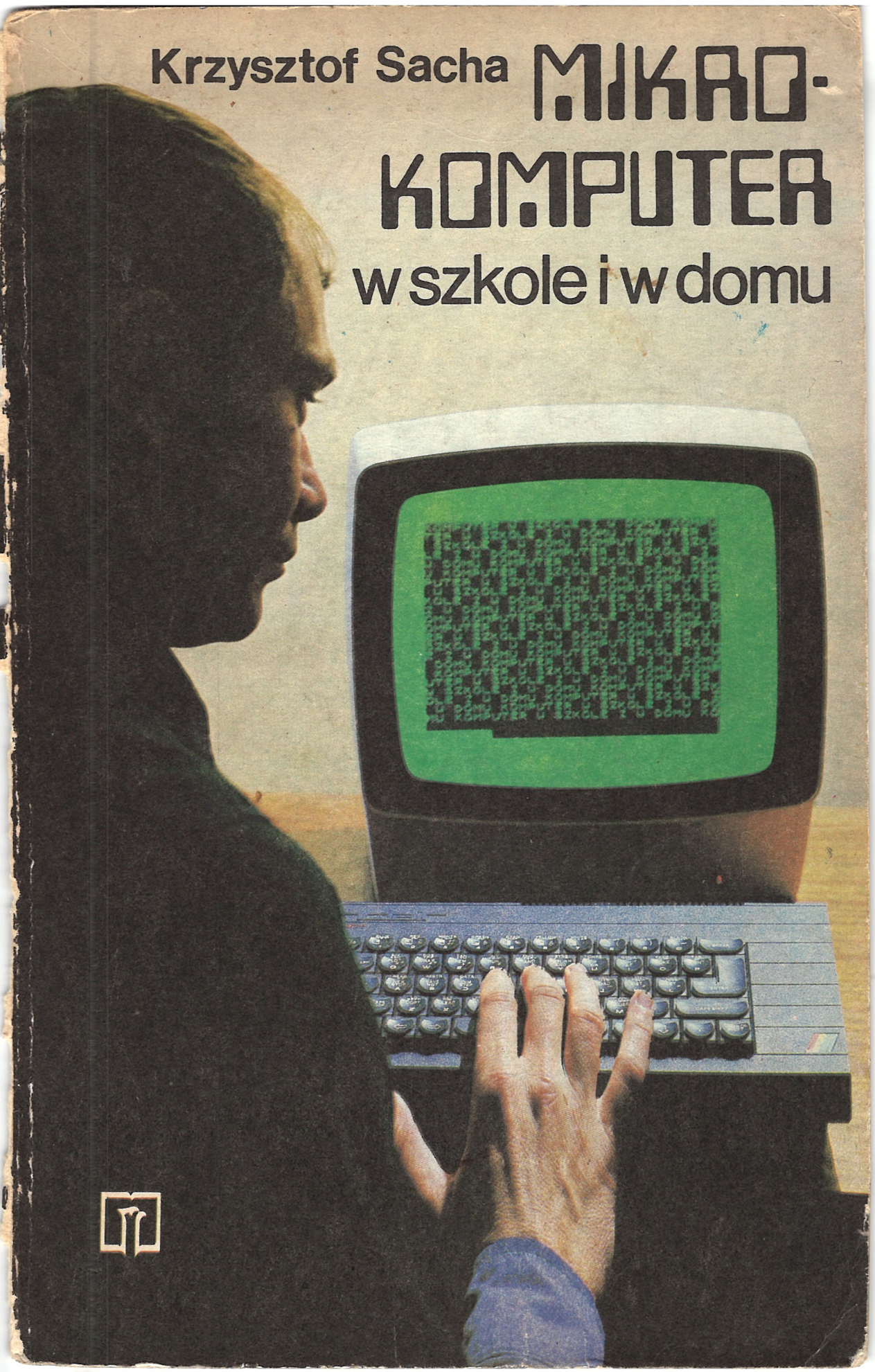
Krzysztof Sacha

MIKRO-KOMPUTER

w szkole i w domu



Krzysztof Sacha

Mikrokomputer w szkole i w domu

Wydanie pierwsze

Warszawa dnia 23 IV 1995

Drogiemu koledze Michałowi
w dniu urodzin
koleżanka
„Przez komputer do serca..."

Warszawa 1988
Wydawnictwa Szkolne i Pedagogiczne

Okładkę projektował: *Jan Sarnecki*
Redaktor: *inż. Hanna Stattler*
Recenzenci: *mgr Witold Kranas* — pracownik naukowy Centrum Astronomicznego, nauczyciel Liceum Ogólnokształcącego
mgr inż. Zbigniew Adamowski — nauczyciel Liceum Ogólnokształcącego im. Stefana Batorego w Warszawie

Redaktor techniczny: *Agnieszka Ziemkiewicz, Teresa Chruścikowska*
Korektor: *Małgorzata Lenk*

Celem książki jest przedstawienie zasad funkcjonowania mikrokomputera oraz możliwości jego wykorzystania w życiu codziennym, szczególnie w dydaktyce szkolnej. Książka nie zakłada żadnego przygotowania specjalistycznego u Czytelnika i nie jest przeznaczona dla ludzi zajmujących się profesjonalnie techniką komputerową. Przeznaczona jest natomiast jako książka pomocnicza dla uczniów liceów ogólnokształcących.
Na jej treść składają się informacje wprowadzające, informacje o programowaniu w języku *Basic* (*Spectrum, Meritum*), proponowane zastosowania w szkole — wykresy funkcji, symulacje eksperymentu fizycznego, gromadzenie danych — oraz opis budowy i programowania mikrokomputera.
Książka pomocnicza dla ucznia

ISBN 83-02-03589-0

© Copyright by Wydawnictwa Szkolne i Pedagogiczne Warszawa 1987

Wydawnictwa Szkolne i Pedagogiczne. Warszawa 1988 Wydanie I
Nakład 49 780 + 220 egz. Ark. wyd. 10,0 Ark. druk. 10,0
Papier tekstowy offsetowy kl. V 70 g rola 61 cm
Oddano do składania 1987-03-27 Podpisano do druku 1987-11-27
Druk ukończono w maju 1988 r.
Wykonanie: Prasowe Zakłady Graficzne
Ciechanów ul. Sienkiewicza 51
Zam. 066257 „MOiW"-15

Spis treści

Wprowadzenie 5

1. Poznajemy nasz mikrokomputer 7

1.1. Uruchamiamy mikrokomputer *Meritum* 7
1.2. Piszemy program 11
1.3. Przechowywanie programów 16
1.4. Uruchamiamy mikrokomputer *Spectrum* 21
1.5. Programujemy i piszemy przy użyciu kolorowych znaków 25

2. Programowanie w języku *Basic* 31

2.1. Zanim zaczniemy programować 31
2.2. Elementy języka 36
2.3. Podstawowe instrukcje 42
2.4. Wprowadzanie danych do programu 50
2.5. Podprogramy. Przykład programu 56
2.6. Niskorozdzielcza grafika mikrokomputera *Meritum* 64
2.7. Grafika mikrokomputera *Spectrum* 68

3. Przykłady programów 74

3.1. Konstrukcje i wykresy funkcji 74
3.2. Komputerowy eksperyment 81
3.3. Rysowanie mapy — komputerowy egzamin 88

4. Budowa i programowanie mikrokomputera 94

4.1. Języki programowania 94
4.2. Operacje arytmetyczne na liczbach dwójkowych 97
4.3. Budowa mikrokomputera 101
4.4. Lista rozkazów mikroprocesora *Z80* 106
4.5. Interpretator języka *Basic* 112

5. Rozwiązania zadań 118

Wprowadzenie

Pierwsze komputery pojawiły się na świecie pod koniec drugiej wojny światowej jako wielkie i kosztowne urządzenia przeznaczone do wykonywania obliczeń naukowo technicznych. Początkowo zakres zastosowań był dość wąski, dopiero z biegiem lat komputery zaczęły wykazywać swą przydatność. Równocześnie dokonywał się stały postęp w ich konstrukcji, czego rezultatem był wszechstronny wzrost możliwości, zmniejszenie rozmiarów i spadek ceny. Na początku lat siedemdziesiątych nastąpił przełom technologiczny: technologia wysokiego stopnia scalenia doprowadziła do pojawienia się mikrokomputerów.

Dzięki bardzo niskiej cenie, nieskomplikowanej obsłudze i małym wymiarom, mikrokomputery znalazły zastosowanie w różnych dziedzinach życia. W przodujących pod względem rozwoju gospodarczego krajach skomputeryzowany jest niemal cały przemysł oraz duża część złożonych wyrobów przemysłowych. Komputerowe układy sterujące nadzorują pracę instalacji przemysłowych, urządzeń medycznych i systemów telekomunikacyjnych. Skomputeryzowany sprzęt biurowy zapewnia lepszą pracę administracji i sprawniejszą obsługę klientów. Komputerowe systemy bankowo-kredytowe zapewniają elastyczne dysponowanie własnym kapitałem, a komputerowe systemy rezerwacji miejsc hotelowych, biletów lotniczych i kolejowych gwarantują szybką i bezbłędną organizację wyjazdów i podróży. Mikrokomputery są szeroko stosowane w sprzęcie powszechnego użytku, jak pralki, maszyny do szycia, magnetowidy. Olbrzymie możliwości kryją się również w zrealizowanych już w niektórych krajach ogólnodostępnych systemach sieci komputerowych podobnych pod względem zasięgu do sieci telefonicznej. Dzięki tej sieci połączeń możliwa jest szybka wymiana informacji, obejmująca zdalny dostęp do bibliotek, połączony z automatycznym wyszukiwaniem potrzebnych danych.

Równocześnie z gwałtownym wzrostem zastosowań w przemyśle, gospodarce i administracji, nieustannie zwiększa się sprzedaż mikrokomputerów dla osób prywatnych. Odpowiednio zaprojektowany i wyposażony w zestaw niezbędnych programów mikrokomputer — nazywany komputerem osobistym lub domowym — stał się towarem powszechnego użytku o cenie porównywalnej z ceną telewizora. Już obecnie co dziesiąty statystyczny Amerykanin ma w swoim domu mikrokomputer, którego używa do zabawy lub do nauki, do prowadzenia domowej rachunkowości, do pracy zawodowej lub do redagowania listów wysyłanych do znajomych. Wraz z upowszechnieniem

komputerów powstała pilna potrzeba upowszechnienia umiejętności posługiwania się tymi urządzeniami. Oczywiście, nie każdy musi umieć zbudować komputer, nie każdy też musi być wykwalifikowanym programistą, szczególnie że większość oprogramowania kupuje się po prostu w sklepie. Każdy natomiast powinien umieć posługiwać się komputerem tak, jak posługuje się telefonem, magnetofonem, samochodem lub pralką automatyczną.

Przyczyną szerokiego rozpowszechnienia komputerów jest ich niezwykła uniwersalność. Komputery są urządzeniami umożliwiającymi rozwiązanie różnych problemów na drodze obliczeń arytmetycznych i logicznych. Wyniki ich pracy mogą być przedstawione w postaci tekstu, rysunku lub dźwięku; mogą też być przekazane wprost do maszyn i urządzeń jako sygnały sterujące. Aby jakiś problem rozwiązać musimy jednak podać komputerowi dokładny opis postępowania prowadzącego do tego celu. Opis taki sporządzamy w postaci **programu**, stanowiącego ciąg instrukcji (poleceń) wykonywanych następnie przez komputer. Zestaw instrukcji, wraz z regułami budowy programu, nazywamy **językiem programowania**. Na ogół możliwe jest programowanie komputera w kilku różnych językach programowania, jednak wśród komputerów powszechnego użytku największą popularność zdobył język *Basic*.

Język Basic został opracowany w 1964 r. na uniwersytecie w Dartmouth, USA, jako *uniwersalny język programowania dla początkujących* (ang. *Beginners All-purpose Symbolic Instruction Code*). Od tego czasu minęło ponad 20 lat, w czasie których wielu ludzi ulepszało ten język tak, że obecnie niemal w każdym komputerze wykorzystuje się inną jego wersję. W przyjętej terminologii wyrażamy ten fakt mówiąc o różnych **implementacjach** języka dostępnych dla różnych typów komputerów. W tej książce przedstawiono implementacje języka *Basic* dostępne dla najbardziej rozpowszechnionych w kraju mikrokomputerów: *ZX Spectrum* i *Meritum*. Niestety, implementacje te różnią się w wielu szczegółach. W związku z tym, w dalszej części książki będziemy omawiać przede wszystkim to, co jest wspólne dla obydwu komputerów. Może się więc zdarzyć, że rzeczywista implementacja języka oferuje coś więcej niż tu napisano. Nie powinno się natomiast zdarzyć tak, że oferuje mniej. W takich przypadkach, gdzie różnice są duże, będzie wyraźnie zaznaczone, o którym komputerze mowa.

Na zakończenie wprowadzenia uwaga metodologiczna. Autor założył, że **Czytelnik ma możliwość posługiwania się komputerem.** Próby uczenia się obsługi i programowania komputera bez dostępu do maszyny stanowią zwykłą stratę czasu. Wszystkie niejasności w tekście mogą być najlepiej i naszybciej wyjaśnione na drodze eksperymentu. Pamiętajmy, że **naciskając dowolne klawisze komputera, pisząc polecenia i instrukcje (być może błędne), nigdy nie uszkodzimy komputera.** W najgorszym razie wprowadzimy go w stan, w którym przestanie reagować na naciskane przez nas klawisze. W każdym takim przypadku możemy zawsze odzyskać kontrolę nad komputerem przez chwilowe wyłączenie napięcia zasilającego, po którym rozpoczniemy pracę od początku.

1
Poznajemy nasz mikrokomputer

W pierwszym rozdziale zapoznamy się ze sposobem uruchamiania i obsługiwania mikrokomputerów *Meritum* i *Spectrum*. Pierwsze dwa podrozdziały dotyczą *Meritum*, dwa ostatnie *Spectrum*. Podrozdział 1.3 dotyczy zagadnień wspólnych dla większości mikrokomputerów z tym jednak, że wszystkie podane tam przykłady poleceń pochodzą z zestawu poleceń realizowanych przez *Meritum*. Ze względu na sposób podania materiału, użytkowników *Spectrum* zachęcam do przeczytania rozdziału w całości.

1.1
Uruchamiamy mikrokomputer Meritum

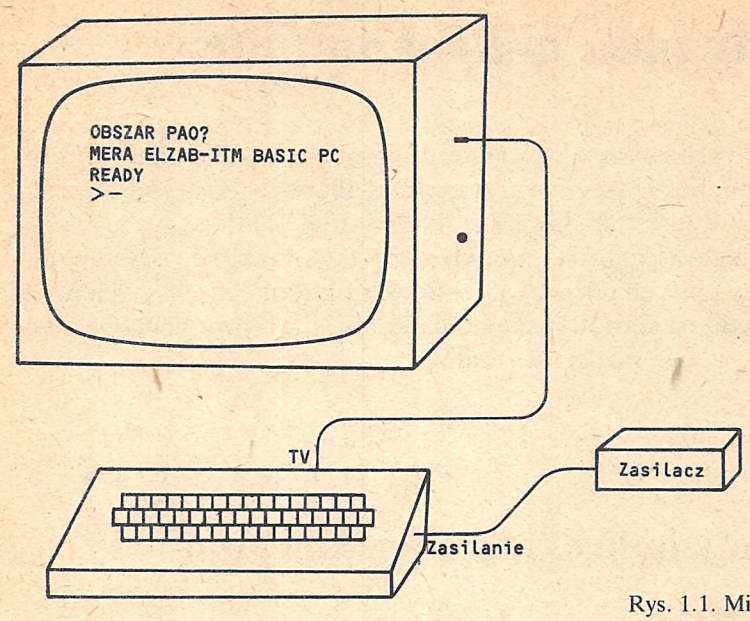
Mikrokomputer *Meritum* mieści się w pudełku wielkości dużej książki. **Po dołączeniu zasilacza i włączeniu napięcia mikrokomputer jest gotów do pracy — już działa.** Klawiatura wskazuje sposób naszego porozumiewania się: wszystkie polecenia i informacje dla mikrokomputera będziemy pisać na klawiaturze. Swoje odpowiedzi mikrokomputer napisze (wyświetli) na ekranie telewizora, połączonego odpowiednim kablem z gniazdem na jego obudowie. Ściśle biorąc, mikrokomputer *Meritum* jest przystosowany do bezpośredniej współpracy z monitorem, czyli telewizorem, z którego usunięto układy wielkiej częstotliwości. Jeżeli nie mamy monitora, możemy wykorzystać dowolny odbiornik telewizyjny, łącząc go z komputerem poprzez specjalny adapter OTV dostarczany na zamówienie przez producenta komputera. Telewizor musimy dostroić do częstotliwości pracy adaptera, znajdującej się w zakresie 3-4 kanału telewizyjnego. Wymienione urządzenia: mikrokomputer z klawiaturą, zasilacz i monitor (lub telewizor z adapterem OTV) tworzą minimalny użyteczny zestaw komputera osobistego (rys. 1.1). I to jest wszystko, co musimy wiedzieć, aby przystąpić do pracy. Zacznijmy od czegoś prostego.

Pierwszym obrazem wyświetlonym na ekranie po włączeniu napięcia jest napis:

OBSZAR PAO? -

stanowiący zapytanie komputera, czy chcemy zarezerwować część pamięci na cele specjalne. Nie chcemy; przyciskamy więc klawisz z napisem *ENTER*, na co komputer odpowiada tekstem

MERA ELZAB - ITM BASIC PC
READY
> –



Rys. 1.1. Mikrokomputer *Meritum*

z którego ważne są jedynie słowa: *BASIC* i *READY*. Pierwsze z nich informuje, że możemy porozumiewać się z komputerem w języku *Basic*, drugie stanowi potwierdzenie gotowości komputera do pracy. Technika komputerowa, w tym mikrokomputery, powstała i rozwinęła się w USA, stąd większość słów i zwrotów używanych w komunikowaniu się z komputerami jest zaczerpnięta z języka angielskiego. Słowo *READY* znaczy po polsku *GOTÓW*. Wyświetlony na ekranie znak - (kreska), nazywany **kursorem**, wskazuje miejsce, w którym pojawią się pisane przez nas polecenia. Jeżeli chcemy, aby komputer coś zrobił, musimy mu bardzo dokładnie powiedzieć, czego oczekujemy. Wydajemy więc polecenie (tzn. piszemy na klawiaturze)

PRINT 5

co po polsku znaczy *DRUKUJ 5*. Na ekranie nic się nie dzieje, dlaczego? Dlatego, że komputer wykona polecenie dopiero wtedy, gdy upewni się, że już napisaliśmy je do końca. **Znakiem zakończenia jest naciśnięcie klawisza** *ENTER* (po polsku: *WPROWADŹ*), **którym musimy kończyć każde polecenie i każdą linię programu.**

Naciskamy więc *ENTER* i na ekranie pojawia się wynik

5

wraz ze znanym już tekstem
READY
> –
sygnalizującym gotowość wykonania dalszych poleceń.

A co będzie, jeżeli się pomylimy? Sprawdźmy. Po napisaniu błędnego polecenia, np.
PRIND 5
(i naciśnięciu klawisza *ENTER*) na ekranie ukaże się napis
? SN ERROR
stanowiący komunikat o błędzie. Słowo *ERROR* znaczy *BŁĄD*, a pozostałe dwie litery określają typ błędu; *SN* oznacza błąd w składni polecenia. Dalsza część wyświetlonego napisu
READY
> –
upewnia nas, że komputer jest gotów przyjąć następne polecenia. Pełna lista kodów błędów jest podana w fabrycznej instrukcji komputera.

Wszystko wygląda prosto, warto jednak zastanowić się przez chwilę, co właściwie komputer zrobił. A wykonał on wiele skomplikowanych czynności. Najpierw **odczytał tekst polecenia, wyświetlając go jednocześnie na ekranie** (klawiatura nie jest połączona z telewizorem — aby znak pojawił się na ekranie, musi zostać wysłany tam przez komputer). Po odebraniu znaku *ENTER* przeanalizował tekst i stwierdził, jakie zawiera polecenie. Dopiero po analizie tekstu i rozpoznaniu polecenia wydruku (*PRINT*) **obliczył wartość argumentu i wyświetlił wynik na ekranie**. Komunikat o wystąpieniu błędu w drugim przykładzie był spowodowany niezrozumiałym dla komputera słowem *PRIND*. Zwróćmy uwagę na zwrot „obliczył wartość argumentu". Aby go wyjaśnić, napiszmy kolejne polecenie
PRINT 5+2
Po naciśnięciu klawisza *ENTER* na ekranie pojawi się wynik
7
A więc komputer nie wyświetla tekstu umieszczonego za słowem kluczowym *PRINT*, lecz rozpoznając tam **wyrażenie arytmetyczne** oblicza jego wartość i wyświetla wynik. Reguły obliczania wartości wyrażeń są zgodne z regułami arytmetyki *:
— najpierw potęgowanie (oznaczone symbolem ↑, np. $3 \uparrow 2$ oznacza 3^2);
— następnie mnożenie (znak *) i dzielenie (znak /);
— na końcu dodawanie (znak +) i odejmowanie (znak –).
Działania w nawiasach są wykonywane w pierwszej kolejności.
Jeżeli więc napiszemy polecenie
*PRINT (4 + 3↑2 – 6 * (8 + 6))/7*
zawierające wyrażenie $(4 + 3^2 - 6(8 + 6))/7$, to na ekranie otrzymamy wynik
–10.1429

---

* Dokładniejszy opis wyrażeń arytmetycznych podano w p. 2.2.

Zwróćmy uwagę, że **do oddzielenia części całkowitej od ułamkowej jest użyta kropka (.) a nie przecinek.** Tę samą konwencję musimy zachować umieszczając niecałkowite liczby w treści poleceń. Użycie przecinka w tej roli nie zostanie przez komputer zrozumiane.

Napiszmy teraz dla porównania polecenie

PRINT "5+2"

Tym razem na ekranie zostanie wyświetlony napis

5+2

Oczywiście po naciśnięciu klawisza *ENTER*; w dalszym ciągu nie będziemy już o tym warunku przypominać. Umieszczenie dowolnego napisu w cudzysłowie powoduje, że jest on traktowany jako tekst, tzn. łańcuch znaków, z których jest zbudowany. Obydwa rodzaje argumentów, tzn. liczby i wyrażenia arytmetyczne oraz teksty mogą być łączone w tym samym poleceniu, np.

PRINT "DODAWANIE:", "5+2 ROWNA SIE"; 5+2

Polecenie to zawiera trzy argumenty:

— dwa teksty: *"DODAWANIE:"* i *"5+2 ROWNA SIE"*

— jedno wyrażenie: *5+2*

Argumenty są oddzielone (odseparowane) znakami: , (przecinek) oraz ; (średnik). Rola tych znaków jest różna:

; (średnik) — powoduje wyświetlenie następnego argumentu bezpośrednio za poprzednim,

, (przecinek) — jest znakiem tabulacji przesuwającym miejsce wyświetlenia następnego argumentu na pozycję 16, 32 lub 48 znaku w wierszu.

Po naciśnięciu klawisza *ENTER* na ekranie pojawi się więc napis

DODAWANIE: 5+2 ROWNA SIE 7

Powróćmy na chwilę do błędów i pomyłek. **Jeżeli zauważymy pomyłkę przed naciśnięciem klawisza *ENTER*, to możemy błąd poprawić.** Naciskając klawisz ze znakiem ← (strzałka w lewo) powodujemy cofanie kursora i kasowanie kolejnych znaków. W miejsce skasowanych możemy wpisać nowe. Naciskając klawisz funkcyjny *SHIFT* i jednocześnie klawisz ze strzałką ← cofamy kursor na początek linii z jednoczesnym skasowaniem wszystkich napisanych znaków.

Podsumowanie

— umiemy uruchomić komputer: wystarczy dołączyć monitor lub telewizor i włączyć zasilanie;

— umiemy wykonywać obliczenia typu kalkulatorowego i wyświetlać teksty: za pomocą polecenia *PRINT*; wiemy też jak budować wyrażenia arytmetyczne;

— wiemy, że każde polecenie kończymy naciskając klawisz z napisem *ENTER*;

— umiemy poprawiać błędnie napisany tekst polecenia: błędy kasujemy naciskając klawisz ← i dopisując tekst poprawny.

Pytania i zadania

1. Jakie urządzenia są niezbędne do uruchomienia mikrokomputera *Meritum*?
2. Które z podanych wyrażeń są poprawne, a które błędne:
   1 (((1 + 2) + 4.1)*5) (1,2+2) 22.3↑2.1/8
3. Napisz polecenia powodujące obliczenie wszystkich wyrażeń z pytania 2.
4. Napisz polecenie, które wyświetli na ekranie: osiem gwiazdek, przerwę o długości równej ośmiu gwiazdkom, osiem gwiazdek.
5. Napisz polecenie powodujące wyświetlenie na ekranie własnego imienia.

1.2
Piszemy program

Wszystkie polecenia pisane przez nas w p. 1.1 były przez komputer natychmiast wykonywane i zapominane. Poza najprostszymi przypadkami nie jest to zbyt dogodne. Kilkakrotne wykonanie tego samego ciągu poleceń wymaga za każdym razem napisania go od nowa. Znacznie dogodniejsza jest na ogół możliwość rozdzielenia etapu wprowadzania poleceń i późniejszego ich wykonania. Zauważmy, że ciąg poleceń dokładnie określa wymagane działanie maszyny. Stanowi on **program** działania komputera. Polecenia wchodzące w skład programu są zwyczajowo nazywane **instrukcjami.**

W języku *Basic* programem jest ciąg instrukcji opatrzonych numerami. Po rozpoczęciu wykonania programu instrukcje są wykonywane kolejno, w kolejności wyznaczonej przez ich numerację. Na przykład program

```
10 PRINT, "  *"
20 PRINT, "  ***"
30 PRINT, "*****"
```

narysuje na ekranie trójkąt równoramienny o podstawie złożonej z pięciu gwiazdek.

W czasie pisania programu każdą linię musimy kończyć naciśnięciem klawisza *ENTER*. Umieszczenie numerów przed instrukcjami informuje komputer, że są to instrukcje programu, które należy zapamiętać, a nie kolejne polecenia, które należy natychmiast wykonać. Wykonanie zapamiętanego programu rozpocznie się dopiero po napisaniu polecenia

```
RUN
```

(oczywiście bez numeru). Polecenie to, jak wszystkie inne polecenia, jest realizowane przez mikrokomputer natychmiast, powoduje więc rozpoczęcie wykonania aktualnie zapamiętanego programu.

Na razie znamy tylko jedną instrukcję języka *Basic*: instrukcję *PRINT*.

Zanim poznamy dalsze, musimy poświęcić chwilę uwagi argumentom instrukcji, którymi mogą być nie tylko liczby, wyrażenia arytmetyczne i teksty, ale również **zmienne**. Pojęcie zmiennej znamy z matematyki: jest to pewien obiekt, któremu możemy nadać dowolną wartość. Rozumienie zmiennej w języku *Basic* jest takie samo. Również forma zapisu zmiennych jest zbliżona do postaci przyjętej w matematyce *, tzn. możemy je nazywać *X*, *Y*, *A*, *A*1 itp. Istotną nowością jest natomiast to, że wartościami niektórych zmiennych mogą być nie liczby, ale teksty. Zmienne, których wartościami są teksty noszą nazwę **zmiennych tekstowych** i są wyróżniane przez dodanie na końcu nazwy znaku *$* (symbol dolara), np. *X$*, *A$*, *Z$*. Nadanie wartości zmiennej może nastąpić w wyniku wykonania różnych instrukcji programu. Poznamy jedną z nich.

Instrukcja *INPUT* — wprowadzania danych, pozwala nadawać zmiennym wartości pisane na klawiaturze w czasie wykonania programu. Argumentami instrukcji *INPUT* mogą być pooddzielane przecinkami nazwy dowolnych zmiennych. W chwili pisania instrukcji wartości, które zostaną nadane zmiennym są jeszcze nieznane. Dla ilustracji napiszemy następujący program działający wyłącznie na tekstach

```
10 PRINT "JAK MASZ NA IMIE?"
20 INPUT X$
30 PRINT "HALO "; X$;", MILO MI CIE POZNAC."
```

Jeżeli mamy przed sobą komputer, to możemy prześledzić sposób wykonania programu. **Wykonanie programu zaczyna się zawsze od linii o najmniejszym numerze** — w naszym przykładzie 10. Znajduje się tu instrukcja *PRINT*, której wykonanie powoduje wyświetlenie na ekranie podanego w cudzysłowie tekstu. **Bezpośrednio po zakończeniu wykonania instrukcji komputer rozpoczyna wykonanie następnej instrukcji, o następnym numerze.** W naszym przypadku jest to instrukcja *INPUT*, której wykonanie przebiega w dwóch etapach. Najpierw komputer wyświetla w nowym wierszu znak zapytania (*?*), aby pokazać, że oczekuje na wprowadzenie danych, potem przechodzi do oczekiwania na napisanie przez nas wartości zmiennej wymienionej w treści instrukcji — w naszym przykładzie *X$*. Wykonanie naszego programu do tego miejsca spowoduje więc wyświetlenie na ekranie fragmentu konwersacji

```
JAK MASZ NA IMIE?
? -
```

Teraz nasza kolej. Podajemy swoje imię, które komputer zapamiętuje jako wartość zmiennej tekstowej *X$*, po czym przechodzi do wykonania ostatniej instrukcji — *PRINT*. W rezultacie, cała konwersacja widoczna na ekranie ma postać

```
JAK MASZ NA IMIE?
? KRZYS
```

---

* Dokładniejszy opis nazw zmiennych zostanie podany w p. 2.2.

HALO KRZYS, MILO MI CIE POZNAC
READY
> –

Ostatnie dwa wiersze zawierają znany nam komunikat kwitujący poprawne zakończenie wykonania programu i sygnalizujący gotowość komputera do przyjęcia nowego polecenia (lub nowego programu).

Poprawne wykonanie programu jest możliwe tylko wtedy, gdy nie zawiera on błędów. W czasie wykonywania programu komputer sprawdza poprawność wszystkich instrukcji i sygnalizuje zauważone błędy komunikatem złożonym z dwuliterowego kodu typu błędu, słowa *ERROR* i numeru wiersza, w którym błąd wystąpił. Po wyświetleniu komunikatu, komputer przechodzi w stan oczekiwania na dalsze polecenia lub linie programu. Wyjątkowym przypadkiem jest wykrycie błędu składniowego. Błędna instrukcja musi zostać poprawiona. komputer przechodzi więc automatycznie do trybu edycji, w którym jest możliwe poprawienie dowolnych znaków w linii programu. Na ekranie pojawia się numer błędnej instrukcji, np.

> RUN
SN ERROR IN 30
30 –

i komputer oczekuje na wprowadzenie poleceń edycyjnych. Opis edytora (podany w fabrycznej instrukcji komputera) jest dość skomplikowany, a jego znajomość nie jest niezbędna. Na kilku dalszych stronach poznamy prostszy sposób zmiany całych linii programu, a informację o edytorze ograniczymy do stwierdzenia, że wyjście z trybu edycji następuje po naciśnięciu klawisza *ENTER*.

Po zakończeniu wykonania program jest nadal przechowywany w pamięci mikrokomputera i może zostać ponownie uruchomiony tym samym poleceniem *RUN*. Zróbmy to nadając zmiennej *X$* np. wartość Iza. Na ekranie powstanie obraz konwersacji

JAK MASZ NA IMIE?
? IZA
HALO IZA, MILO MI CIE POZNAC

zakończony tym samym komunikatem stwierdzającym zakończenie wykonywania programu.

Powróćmy na chwilę do etapu pisania programu. Każda instrukcja ma tu swój numer, przy czym w naszym programie numery zmieniają się nie co 1, lecz co 10. Świadczy to o naszej przezorności, gdyż dzięki temu możemy łatwo dopisywać nowe instrukcje i wstawiać je przed, lub pomiędzy już istniejące. Na przykład, możemy na początku konwersacji przedstawić nasz komputer. W tym celu napiszmy instrukcję

5 PRINT "DZIEN DOBRY, TU KOMPUTER MERITUM"

Po odczytaniu instrukcji mikrokomputer przejrzy cały zapamiętany program, znajdzie miejsce, w którym według kolejności numeracji powinna znaleźć się nowa linia i wstawi tam napisaną instrukcję. W rezultacie ostatnio na-

pisana instrukcja znajdzie się na początku programu. Możemy to łatwo sprawdzić pisząc polecenie
LIST
które powoduje wyświetlenie, czyli wylistowanie na ekranie całego zapamiętanego programu
5 PRINT "DZIEN DOBRY, TU KOMPUTER MERITUM"
10 PRINT "JAK MASZ NA IMIE?"
20 INPUT X$
30 PRINT "HALO "; X$;" , MILO MI CIE POZNAC"
W ten sam sposób możemy dodać linie o numerach np. *11* lub *28*, które zostaną wstawione w miejsca wskazywane przez ich numerację. A co będzie, jeżeli oznaczymy linię, którą chcemy wpisać, numerem już istniejącej linii programu, np.
30 PRINT X$; " ? A CO TO ZA POGANSKIE IMIE?"
Komputer zrobi to samo co poprzednio — wstawi nową linię do programu w miejsce wynikające z numeracji. Tym samym „stara" linia z numerem *30* zostanie zniszczona. Jeżeli masz dostęp do komputera, możesz to natychmiast sprawdzić za pomocą polecenia *LIST*, które spowoduje wyświetlenie na ekranie nowego tekstu programu ze zmienioną linią *30*. W taki sam sposób możemy zmieniać i inne linie programu. Możemy też zbędne linie kasować, pisząc sam numer,np. *20*
Jest to tzw. pusta linia, która zastępuje „starą" linię z numerem *20*, powodując jej usunięcie.

Możliwość dopisywania i wymiany linii będziemy wykorzystywać do wprowadzania zmian i poprawek do napisanych programów. Zanim jednak zmodyfikujemy nasz program, zapoznajmy się z nieco inną postacią instrukcji *INPUT*. W rozszerzonej wersji tej instrukcji jako pierwszy argument podajemy tekst umieszczony w cudzysłowie i oddzielony od dalszych argumentów średnikiem. W chwili wykonania instrukcji komputer wyświetli cały tekst, kończąc go znakiem zapytania (?), po czym przejdzie do oczekiwania na wprowadzenie danych. Dopiszmy do naszego programu
10 INPUT "JAK MASZ NA IMIE?"; X$
Po uruchomieniu programu poleceniem *RUN* na ekranie pojawi się pytanie
DZIEN DOBRY, TU KOMPUTER MERITUM.
JAK MASZ NA IMIE? –
Tak jak w poprzednich przykładach możemy teraz podać swoje imię i otrzymać uprzejmą odpowiedź komputera.

Odpowiednie numerowanie linii programu może być prowadzone automatycznie przez komputer. Polecenie *AUTO*, np.
AUTO 50, 10
rozpoczyna numerowanie linii przez komputer poczynając od numeru podanego jako pierwszy parametr polecenia (tu *50*) ze zwiększeniem numeru co linia o wartość podaną jako drugi parametr (tu *10*). Powrót do trybu bezpośre-

dniego wykonywania poleceń wymaga naciśnięcia klawisza *BREAK* (*przerwij*). Na ekranie pojawi się

READY

> –

sygnalizując gotowość do przyjęcia kolejnych poleceń lub dowolnie numerowanych instrukcji.

W czasie listowania programu (polecenie *LIST*) zawartość ekranu jest przesuwana do góry z usuwaniem szczytowych linii. Szybkość przesuwania uniemożliwia obejrzenie programu, który nie mieści się w całości na ekranie. W tej sytuacji konieczne jest ograniczenie w poleceniu *LIST* zakresu wyświetlania przez podanie numerów linii, które mają być wyświetlone. Na przykład

LIST 100—150

spowoduje wyświetlenie na ekranie linii o numerach z zakresu *100...150*. Przerwanie listowania programu jest zawsze możliwe przez naciśnięcie klawisza *BREAK*.

Podsumowanie

— wiemy, jak pisać program: jako ciąg numerowanych instrukcji, przy czym numerację prowadzimy sami albo prowadzi ją automatycznie mikrokomputer po otrzymaniu polecenia *AUTO*;
— wiemy, jak usuwać, zmieniać lub dodawać instrukcje programu: pisząc nowe z odpowiednią numeracją;
— umiemy program wylistować: poleceniem *LIST*, wiemy, jak ograniczyć zakres listowania i jak je przerwać: poleceniem *BREAK*;
— umiemy uruchomić program: poleceniem *RUN*;
— znamy dwie instrukcje języka *Basic*: *INPUT* i *PRINT*.

Pytania i zadania

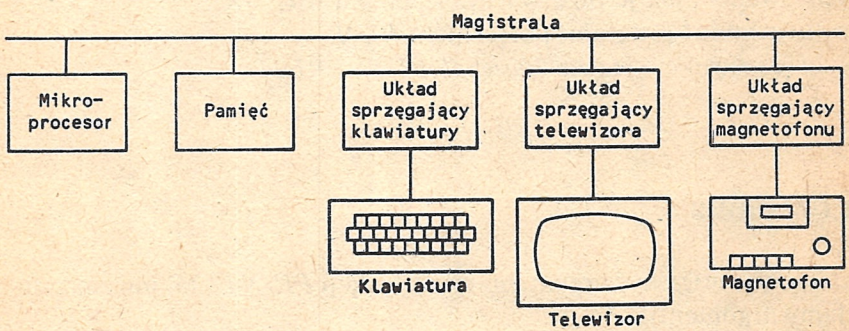
1 Jakie elementy mogą być argumentami instrukcji *PRINT*? Jakimi znakami oddzielamy argumenty?
2 Napisz program, który odczyta wartość dwóch liczb wprowadzonych z klawiatury i wyświetli wartość sumy tych liczb.
3 Jakie elementy mogą być argumentami instrukcji *INPUT*? Jakimi znakami oddzielamy argumenty?
4 Opisz sposób wykonania programu

10 INPUT "IMIE I WIEK;X$;Y
20 PRINT X$;"MA";Y;" LAT"

5. W jaki sposób usunąć zbędne linie programu?
6. Do czego służą polecenia *AUTO* i *LIST*?
7. Czy można kilkakrotnie wykonać ten sam program? Jakiego polecenia trzeba użyć dla uruchomienia programu?

1.3 Przechowywanie programów

Centralnym elementem mikrokomputera (rys. 1.2) jest **mikroprocesor**. Jest to niewielkie urządzenie zdolne do wykonywania dowolnych ciągów instrukcji tworzących program jego działania. Zarówno program jak i wszystkie dane oraz otrzymane wyniki są przechowywane w **pamięci**, stanowiącej drugi istotny element mikrokomputera. W czasie pracy mikroprocesor nieustannie pobiera z pamięci instrukcje i dane potrzebne do ich wykonania, a następnie zapisuje w pamięci otrzymane wyniki. Pamięć wykorzystywana w taki sposób nosi nazwę **pamięci operacyjnej**. Pamięć operacyjna jest połączona z mikroprocesorem zespołem przewodów tworzących tzw. **magistralę** mikrokomputera. Magistrala stanowi główne łącze, po którym przepływa cała informacja magazynowana i przetwarzana wewnątrz komputera. Trzecim elementem mikrokomputera jest klawiatura. Jest ona dołączona do magistrali za pośrednictwem **układu sprzęgającego**. W podobny sposób (za pośrednictwem specjalnego układu sprzęgającego) jest dołączony do magistrali monitor lub telewizor.



Rys. 1.2. Podstawowe elementy mikrokomputera

Obszar pamięci operacyjnej wykorzystywany przez programy napisane w języku Basic jest podzielony funkcjonalnie na szereg pól. Z naszego punktu widzenia najistotniejsze są trzy:
— pole pamięci programu,
— pole pamięci zmiennych,
— pole pamięci ekranu.

Bezpośrednio po włączeniu zasilania wszystkie trzy pola pamięci są puste. Napisanie programu powoduje umieszczenie go w polu pamięci programu i zarezerwowanie w polu pamięci zmiennych miejsca na wszystkie zmienne występujące w tym programie. Polecenie *RUN* powoduje wykonanie dwóch czynności:

— wyzerowanie pola pamięci zmiennych,
— rozpoczęcie wykonywania instrukcji programu od początku.

Po zakończeniu wykonywania programu zawartość obydwu pól pamięci nie jest kasowana, lecz pozostaje w takim stanie, jaki powstał po wykonaniu ostatniej instrukcji.

Program jest zapisywany w pamięci wraz z numerami instrukcji. Dlatego możliwe jest również rozpoczęcie wykonywania programu od linii o określonym numerze. Numer ten podajemy jako argument w poleceniu *RUN*, np. polecenie

RUN 30

spowoduje wyzerowanie całego pola pamięci zmiennych i rozpoczęcie wykonywania programu od instrukcji oznaczonej numerem *30*. Podobny efekt osiągniemy pisząc polecenie

GOTO 30

(co znaczy *"idź do 30"*) z tą różnicą, że nie nastąpi tu wyzerowanie pola pamięci zmiennych. Jedyną wykonaną czynnością będzie przejście do wykonywania programu poczynając od instrukcji oznaczonej numerem *30*.

Rozpoczęcie wprowadzania nowego programu zmienia jedynie zawartość tych linii poprzedniego programu, których numery pokrywają się z numerami nowo wprowadzanych linii. Może to prowadzić do „zaśmiecenia" programu resztkami poprzednich programów. Aby tego uniknąć możemy postąpić dwojako:

albo zacząć numerację linii nowego programu od numeru wyższego, np. o 100 od ostatniego w starym programie,

albo wyzerować (opróżnić) pole pamięci programu poleceniem

NEW

(co znaczy *nowy*), które zeruje jednocześnie pole pamięci zmiennych.

Wyzerowanie pola pamięci zmiennych następuje również jako efekt wykonania polecenia

CLEAR

(co znaczy *wyczyść*), które nie zmienia jednak zawartości pola pamięci programu. W mikrokomputerze *Meritum* polecenie *CLEAR* wydziela dodatkowo w polu pamięci zmiennych miejsce przewidziane na przechowywanie zmiennych tekstowych. Standardowy przydział (*CLEAR* bez argumentu) wynosi 50 znaków. Wykonanie polecenia, np.

CLEAR 100

spowoduje zarezerwowanie miejsca wystarczającego do zapamiętania 100 znaków tekstu. Zmiany rezerwacji miejsca przeznaczonego na zmienne tekstowe dokonujemy zazwyczaj po komunikacie błędu

OS ERROR
sygnalizującym wyczerpanie dotychczas zarezerwowanego na ten cel miejsca.

Pole pamięci ekranu służy do przechowywania odpowiednio zakodowanej informacji określającej obraz aktualnie wyświetlany na ekranie telewizyjnym. Wyzerowanie pola pamięci ekranu poleceniem

CLS

powoduje wymazanie obrazu z ekranu.

Istotną cechą elementów elektronicznych, z których jest wykonana pamięć mikrokomputera jest tzw. ulotność. To znaczy **informacja jest przechowywana tylko przy włączonym napięciu zasilającym. Wyłączenie napięcia zasilającego powoduje natychmiastowe skasowanie całej zawartości pamięci.** Łatwo to sprawdzić wprowadzając do pamięci jakiś program, odłączając na chwilę zasilacz i po ponownym dołączeniu zasilania pisząc polecenie *RUN*. Żaden program nie zostanie wykonany. Również próba wykonania polecenia *LIST* wykaże brak jakiegokolwiek programu w pamięci. Ta właściwość pamięci mikrokomputera jest bardzo niedogodna w praktyce. Do niektórych programów chcielibyśmy mieć stały dostęp, a trudno przecież wpisywać je wciąż od nowa z klawiatury. Rozwiązaniem problemu jest wyposażenie mikrokomputera w dodatkową pamięć trwałą, która przechowuje informację również przy wyłączonym zasilaniu. Najtańszym rodzajem pamięci o tej właściwości są kasety magnetofonowe. Nagranie piosenki to nic innego jak trwałe zapamiętanie jej tekstu i melodii. W taki sam sposób można przechowywać w kasetach teksty programów i wartości danych. Dołączenie magnetofonu do mikrokomputera wymaga oczywiście specjalnego układu sprzęgającego. Mikrokomputer *Meritum*, podobnie jak większość innych komputerów domowych, jest standardowo wyposażony w układ sprzęgający przystosowany do współpracy z magnetofonem kasetowym. W przypadku mikrokomputera *Meritum* może to być dowolny magnetofon, choć pożądane jest, aby był wyposażony w licznik taśmy, który bardzo ułatwia odszukanie na taśmie potrzebnego programu.

Kasety magnetofonowe okazały się dogodnym nośnikiem informacji: są tanie, a jednocześnie bardzo pojemne. Stąd większość programów przenoszonych między komputerami takimi jak *Meritum* lub *Spectrum* jest przenoszona właśnie na kasetach. Niestety, na ogół przenoszenie programów możliwe jest tylko między komputerami tego samego typu.

Zauważmy, że jakkolwiek taśma magnetofonowa pełni rolę pamięci komputera, to jednak jest to pamięć innego rodzaju niż pamięć operacyjna. **Procesor nie może pobierać i wykonywać instrukcji wprost z magnetofonu.** Przed wykonaniem zapisanego na taśmie programu, program ten musi być przepisany do pamięci operacyjnej. Taśma magnetofonowa pełni rolę **pamięci zewnętrznej** komputera. Do przesyłania programu między pamięcią operacyjną a zewnętrzną służą specjalne polecenia *CSAVE* i *CLOAD*.

Polecenie *CSAVE* (od ang. *SAVE — zabezpiecz*) powoduje zapisanie całego programu na taśmie magnetofonowej i nadanie mu nazwy podanej w cu-

dzysłowie za słowem kluczowym *CSAVE*. **Przed naciśnięciem** klawisza *ENTER* kończącego polecenie należy włączyć magnetofon nastawiony na nagrywanie. Należy też zwrócić uwagę, aby nie próbować zapisu na początkowym odcinku rozbiegówki.

Polecenie *CLOAD* (od ang. *LOAD — ładuj*), którego argumentem jest podana w cudzysłowie nazwa programu, powoduje wyszukanie programu na taśmie, odczytanie go i zapisanie w pamięci operacyjnej. Magnetofon, nastawiony na odtwarzanie, należy włączyć dopiero **po naciśnięciu** klawisza *ENTER* kończącego polecenie. Inną rolę pełni polecenie *CLOAD?* wykonane po operacji *CSAVE*. Polecenie to również powoduje wyszukanie i odczytanie programu z taśmy, nie ładuje go jednak do pamięci, lecz porównuje z jej zawartością. Ewentualna niezgodność jest kwitowana komunikatem *BAD* (*źle*) sygnalizującym błąd w operacji zapisu.

Sprawdźmy działanie nowo poznanych poleceń. Mamy jeszcze w pamięci operacyjnej ostatni program z p. 1.2, a jeżeli nie, to wprowadźmy jeden z podanych tam programów. Przewijamy taśmę do początku, odwijamy rozbiegówkę, piszemy na klawiaturze polecenie

CSAVE "A"

włączamy magnetofon nastawiony na „zapis" i dopiero teraz naciskamy klawisz *ENTER*. Po zakończeniu operacji, co jest potwierdzane przez mikrokomputer komunikatem

READY

>–

przewijamy taśmę do początku, piszemy polecenie kontroli

CLOAD? "A"

naciskamy klawisz *ENTER* i włączamy magnetofon na „odczyt". W chwili wyszukania przez mikrokomputer programu na taśmie, w prawym górnym rogu ekranu pojawiają się dwie gwiazdki (**). W czasie trwania odczytu druga gwiazdka miga. Zasygnalizowanie przez mikrokomputer błędu (komunikat *BAD*) oznacza dla nas konieczność ponownego zapisania programu, najlepiej na innym odcinku taśmy. Zakończenie kontroli bez wykrycia błędu

READY

>–

upewnia nas o prawidłowym zapisaniu programu na taśmie. Możemy teraz spokojnie wyłączyć mikrokomputer wiedząc, że w każdej chwili możemy program załadować od nowa. W tym celu przewijamy taśmę, piszemy polecenie

CLOAD "A"

naciskamy *ENTER* i włączamy magnetofon nastawiony na odtwarzanie. Jeżeli na taśmie jest zapisanych kilka różnych programów, mikrokomputer rozpoczyna poszukiwanie programu o podanej w poleceniu nazwie. W chwili wykrycia początku jakiegoś programu na ekranie pojawiają się dwie gwiazdki (**). W razie niezgodności nazwy program jest pomijany, a na ekranie w miejscu pierwszej gwiazdki jest wyświetlana pierwsza litera jego nazwy. W

czasie ładowania programu druga gwiazdka miga. Prawidłowy odczyt programu kończy się komunikatem
READY
>–
Polecenie *RUN* uruchamia wykonanie programu. Nazwy używane w poleceniach *CSAVE*, *CLOAD?*, *CLOAD* mogą być co najwyżej sześcioznakowe, z tym jednak, że **mikrokomputer rozpoznaje tylko pierwszą literę**. Należy o tym pamiętać, gdyż inaczej może nastąpić pomylenie nagranych na taśmie programów.

Podsumowanie

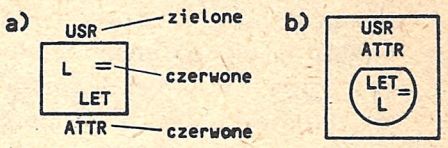
— znamy podstawowe elementy mikrokomputera: mikroprocesor, pamięć operacyjną, urządzenia wejścia-wyjścia (klawiatura i telewizor) i pamięć zewnętrzną (magnetofon);
— znamy podział pamięci operacyjnej i umiemy posługiwać się poleceniami opróżniania różnych pól pamięci: *NEW*, *CLEAR*, *CLS* i uruchamiania programu: *RUN*, *GOTO*;
— wiemy, do czego służy magnetofon, i umiemy go obsługiwać: polecenia *CSAVE*, *CLOAD?*, *CLOAD*.

Pytania i zadania

1. Czym się różni pamięć operacyjna od pamięci zewnętrznej?
2. Wyjaśnij działanie poleceń *NEW*, *CLEAR* i *CLS*.
3. W czasie wykonania pewnego programu pojawił się komunikat
   OS ERROR
   jak usunąć zasygnalizowany błąd?
4. Czym różnią się polecenia: *RUN 50* i *GOTO 50*?
5. Napisz program z pytania 4 w p. 1.2, po czym:
   a) zapisz go na taśmie magnetofonowej,
   b) sprawdź poprawność zapisania,
   c) dopisz do programu jakąś instrukcję, np.
      15 PRINT "ZROZUMIALEM"
      i spróbuj teraz wykonać polecenie *CLOAD?*,
   d) załaduj program z kasety i wykonaj go.

1.4
Uruchamiamy mikrokomputer Spectrum

Mikrokomputer *ZX Spectrum* wygląda podobnie jak *Meritum*. Nieco mniejsze pudełko z klawiaturą na wierzchu i kilkoma gniazdami do dołączenia zasilacza (+9 V) oraz urządzeń zewnętrznych: przede wszystkim telewizora i magnetofonu, ale również drukarki i różnych innych urządzeń specjalnych. W odróżnieniu od *Meritum*, mikrokomputer *Spectrum* jest przystosowany do współpracy z telewizorem kolorowym. Sygnał wyprowadzony przez gniazdo antenowe oznaczone *TV* jest pełnym sygnałem wizyjnym telewizji kolorowej systemu *PAL*. Ponieważ w Polsce przyjęty został inny system kodowania sygnału telewizji kolorowej (*SECAM*), większość dostępnych w kraju odbiorników kolorowych nie będzie mogła współpracować z tym mikrokomputerem. Na szczęście przekazywanie obrazu czarno-białego nie zależy od sposobu kodowania koloru. Wyjście *TV* mikrokomputera *Spectrum* możemy więc bez obawy połączyć kablem z wejściem antenowym dowolnego telewizora czarno-białego. Telewizor powinien zostać dostrojony do częstotliwości 36 kanału. Pierwszą różnicą zauważalną dla początkującego użytkownika jest zupełnie inna niż w *Meritum* budowa klawiatury. Ma mniej klawiszy, za to na każdym widnieje po kilka symboli (rys. 1.3). W modelu *Spectrum 48K* różne grupy symboli są oznaczone różnymi kolorami, co ułatwia zapamiętanie reguł posługiwania się klawiaturą. W modelu *Spectrum Plus* (nowsza wersja *Spectrum*) zrezygnowano z kolorowych oznaczeń, dodając za to kilka klawiszy zawierających najczęściej używane znaki.



Rys. 1.3. Typowy klawisz mikrokomputera *Spectrum*: *a) 48K; b) plus*

Znaczenie każdego klawisza zależy od dwóch czynników: od trybu pracy klawiatury oraz od jednoczesnego przyciśnięcia (lub nie przyciśnięcia) specjalnych klawiszy *CAPS SHIFT* i *SYMBOL SHIFT*. Obydwa klawisze nie odpowiadają żadnym literom ani symbolom, a jedynie służą do zmiany znaczenia pozostałych klawiszy. Co do trybu pracy klawiatury, to czasem zmienia się on automatycznie, a czasem po naciśnięciu odpowiednich klawiszy. Najważniejsze, że zawsze wiadomo, w jakim trybie aktualnie pracujemy. Wskazuje na to rodzaj znaku, wyświetlanego na ekranie w roli kursora. Jak pamiętamy, w *Meritum* kursor miał zawsze postać znaku – (kreska). W *Spectrum* jest inaczej. Kursorem jest migający znak (litera), który wskazuje miejsce wyświetlania znaków na ekranie, a jednocześnie informuje o trybie pracy klawiatury. I tak, kursorem mogą być znaki:

K (od *Keyword — słowo kluczowe*). Po naciśnięciu wybranego klawisza do komputera zostanie wprowadzone słowo kluczowe wypisane w głównym polu klawisza (*LET* na rys. 1.3). Słowa kluczowe odpowiadają instrukcjom lub poleceniom języka *Basic*. W przypadku klawiszy oznaczonych cyframi (*0...9*), które nie mają w głównym polu słów kluczowych, do mikrokomputera zostaje oczywiście wprowadzona odpowiednia cyfra. Przejście do trybu *K* następuje automatycznie, gdy w trakcie pisania programu mikrokomputer oczekuje na wprowadzenie instrukcji lub polecenia.

L (od *Letter — litera*). Po naciśnięciu wybranego klawisza do mikrokomputera zostanie wprowadzona litera lub cyfra, wypisane w głównym polu klawisza (*L* na rys. 1.3). Jednoczesne naciśnięcie klawisza *CAPS SHIFT* i wybranego klawisza wprowadza do mikrokomputera wielką literę. Natomiast jednoczesne naciśnięcie klawisza specjalnego *SYMBOL SHIFT* i wybranego klawisza wprowadza do mikrokomputera symbol relacji lub dodatkowe słowo kluczowe umieszczone w głównym polu klawisza (znak = na rys. 1.3). Tryb *L* jest podstawowym trybem pracy klawiatury, w który wchodzi ona automatycznie po napisaniu słowa kluczowego instrukcji lub polecenia.

C (od *Capital letter — wielka litera*). Jest to odmiana trybu *L*, w której do mikrokomputera są wprowadzane zawsze wielkie litery zamiast małych. Przejściem z trybu *L* do trybu *C* (i odwrotnie) steruje klawisz *CAPS LOCK*.

E (od *Extended — rozszerzony*). Przyciśnięcie wybranego klawisza powoduje wprowadzenie do mikrokomputera słowa kluczowego wypisanego w górnym rzędzie poza głównym polem klawisza (*USR* na rys. 1.3). Jednoczesne naciśnięcie klawisza *SYMBOL SHIFT* i wybranego klawisza wprowadza do mikrokomputera słowo kluczowe wypisane w dolnym rzędzie poza głównym polem klawisza (*ATTR* na rys. 1.3). Przejście klawiatury do pracy w trybie *E* następuje po jednoczesnym naciśnięciu klawiszy *CAPS SHIFT* i *SYMBOL SHIFT* w modelu *Spectrum 48K* lub po naciśnięciu klawisza *EXTEND MODE* w modelu *Spectrum Plus*. Po wprowadzeniu wybranego słowa kluczowego klawiatura wraca do pracy w trybie *L* lub *C*.

G (od *Graphic — graficzny*). Przyciskając wybrane klawisze wyświetlamy na ekranie znaki graficzne widoczne na klawiszach oznaczonych cyframi (*1...8*) albo inne znaki graficzne zdefiniowane przez nas uprzednio i zapisane w pamięci mikrokomputera. Przejście do pracy w trybie *G* następuje po jednoczesnym naciśnięciu klawisza *CAPS SHIFT* i klawisza oznaczonego cyfrą *9* w modelu *Spectrum 48K* lub po naciśnięciu klawisza *GRAPH* w modelu *Spectrum Plus*. Powrót do pracy w trybie *L* lub *C* następuje po naciśnięciu jednego z dwóch klawiszy oznaczonych *9* lub *DELETE*.

I to już prawie wszystko o klawiaturze. Prawie, bo potrzebne jest jeszcze kilka uwag ważnych dla użytkowników modelu *Spectrum 48K* (bez plusa). Jak się

wkrótce okaże, niezwykle ważne przy pisaniu programów są funkcje realizowane przez klawisze oznaczone symbolami:
EDIT, ◊, ◊, ◇, ◇, *DELETE*.

W modelu *Spectrum Plus* klawiatura posiada specjalne klawisze oznaczone tylko tymi symbolami. W modelu *Spectrum 48K* symbole te są umieszczone nad klawiszami oznaczonymi cyframi *1*, *5*, *6*, *7*, *8*, *0* i są osiągalne przez jednoczesne naciśnięcie klawisza *CAPS SHIFT* i odpowiedniego klawisza cyfrowego. Szczególną funkcję pełni również polecenie *BREAK*, posiadające własny klawisz w modelu *Spectrum Plus* i dzielące klawisz ze spacją (*SPACE*) w *Spectrum 48K*, gdzie *BREAK* jest osiągalne przy wciśniętym klawiszu *CAPS SHIFT*.

Tym razem to już naprawdę wszystko. Przypomnijmy, że naciskając dowolne klawisze nigdy nie uszkodzimy mikrokomputera. W najgroszym razie wprowadzimy go w stan, w którym przestanie reagować na nasze polecenia. Jednak wyłączenie na chwilę zasilania i włączenie go z powrotem przywoła go natychmiast do porządku. A więc zaczynamy.

Włączamy mikrokomputer i spoglądamy na ekran. Bezpośrednio po włączeniu zasilania na ekranie miga jakiś obraz, po czym wyświetla się napis
© 1982 Sinclair Research Ltd
Kursora nie widać. Aby go wywołać, naciskamy klawisz *ENTER*, po czym napis znika, a zamiast niego pojawia się migająca litera *K*. Klawisz *ENTER* pełni w *Spectrum* tę samą rolę co w *Meritum*, a więc każde polecenie i każdą linię programu musimy kończyć naciśnięciem tego klawisza. Kursor *K* informuje nas, że komputer oczekuje na wprowadzenie polecenia lub instrukcji. Przyciskamy na próbę klawisz *P* (zawierający też słowo kluczowe *PRINT*) i na ekranie wyświetla się całe słowo *PRINT*, a kursor zmienia się z *K* na *L*, co przypomina nam, że powinniśmy podać argumenty polecenia, które składają się na ogół z liter i cyfr. Naciskamy kolejno *2* i *ENTER*. Na ekranie wyświetla się cyfra *2*, znika treść polecenia i pojawia się napis *0 OK*, *0:1*. A więc działa. Zastanawiamy się tylko dlaczego *2* znalazło się u góry ekranu, jakiś dziwny napis pojawił się u dołu, a kursor i treść polecenia znikły w ogóle. Wynika to z przyjętej w *Spectrum* zasady „gospodarowania" powierzchnią ekranu.

Cały ekran jest podzielony na 24 linie po 32 znaki w linii. Górna część ekranu, zawierająca 22 linie, jest przeznaczona na wyświetlenie treści oraz efektów wykonania programu. Dolna część, zawierająca 2 linie, jest przeznaczona na konwersację użytkownika z mikrokomputerem. Tu są wyświetlane komunikaty o błędach, tu piszemy polecenia lub wiersze programu i tu możemy dokonać edycji pisanego tekstu. Napis *0 OK*, *0:1* jest komunikatem stwierdzającym poprawne wykonanie polecenia. Pierwszy znak określa typ błędu, przy czym *0* to brak błędu, dalszy tekst zawiera objaśnienie błędu w języku angielskim, przy czym *OK* znaczy, że wszystko w porządku, a liczby po przecinku określają numer linii, w której błąd wystąpił, przy czym *0:1* w naszym przypadku nic nie znaczy. Brak kursora nie powinien nas peszyć — na począ-

tku też go nie było. Robimy to samo, co wtedy: naciskamy klawisz *ENTER* i wszystkie napisy znikają, a na dole pojawia się kursor K.

Spróbujmy, czy można napisać polecenie — np. *PRINT* nie korzystając ze słów kluczowych klawiatury, lecz wprowadzając pięć liter: *P,R,I,N,T*. Nie zważając na kursor, a jeszcze lepiej nie patrząc na ekran przyciskamy kolejno klawisze: *P,R,I,N,T,2*. Na ekranie pojawia się napis

PRINT rint2[L]

Nietrudno zgadnąć, co się stało. Przyciśnięcie klawisza *P* w trybie *K* wprowadziło do mikrokomputera słowo kluczowe *PRINT* i spowodowało przejście klawiatury do pracy w trybie *L*. Naciskając pozostałe klawisze wprowadziliśmy tekst *rint2*. Całość nie ma sensu i powinna zostać poprawiona. W tym celu użyjemy klawisza *DELETE* (jeżeli działasz na *Spectrum 48K*, odwróć parę kartek i przypomnij sobie, jak osiągnąć *DELETE*). Naciskamy. Znika *2*, a kursor cofa się o jeden znak do tyłu

PRINT rint [L]

Naciskamy *DELETE* jeszcze cztery razy i na ekranie pozostaje

PRINT [L]

Teraz dopisujemy *2*, naciskamy *ENTER* i na ekranie wyświetla się znane nam już *2* i znany napis *0 OK, 0:1*.

A czy można było tak wprowadzić poprawkę, aby nie kasować potrzebnej cyfry *2*, a tylko usunąć zbędne *rint* ze środka? Można. Napiszmy ponownie

PRINT rint2[L]

i naciśnij klawisz oznaczony symbolem ◊ (z *CAPS SHIFT* w *Spectrum 48K*). Kursor przesunie się pomiędzy *t* a *2*

PRINT rint[L]*2*

Naciskając teraz czterokrotnie *DELETE* usuwamy kolejno cztery znaki przed kursorem do postaci

PRINT [L]*2*

Jeżeli naciśniemy *DELETE* jeszcze raz, to zniknie całe *PRINT* i zostanie tylko

[K]*2*

Nie przejmujmy się. Kursor zmienił się na *K*, więc przyciskając klawisz *P* wprowadzimy z powrotem słowo kluczowe *PRINT*

PRINT [L]*2*

Możemy jeszcze dopisać np. *15+* i po naciśnięciu *ENTER* uzyskać wynik

17

wraz z komunikatem *0 OK, 0:1*. Klawisz ◊ ma również bliźniaka ◊, który przesuwa kursor o jeden znak w prawo.

Skasowanie całej linii, zwłaszcza jeżeli jest ona długa, przez pojedyncze naciskanie klawisza *DELETE* może być uciążliwe. Jeżeli jednak przytrzymamy ten klawisz naciśnięty przez 3–4 sekundy, to zacznie on samoczynnie kasować dalsze litery. Nietrudno sprawdzić, że przyciśnięcie każdego klawisza powoduje takie samo powtarzanie jego funkcji. Najbardziej użyteczne jest to jednak w przypadku klawisza *DELETE* i klawiszy sterujących ruchem kursora ◊ i ◊.

Ostatni eksperyment, jaki zrobimy, to próba wykonania polecenia
PRINT rint2
Naciskamy klawisz *ENTER* i na ekranie pojawia się komunikat
2 variable not found, 0:1
co znaczy: *błąd numer 2, zmienna nie znaleziona*. Ta zmienna to oczywiście *rint2*.

Podsumowanie

— umiemy uruchomić mikrokomputer *Spectrum* i posługiwać się jego klawiaturą: musimy tylko uważać na rodzaj kursora i pamiętać o różnym znaczeniu klawiszy w różnych trybach pracy klawiatury;
— umiemy wprowadzać polecenia i poprawiać błędy: kursor przesuwamy klawiszami ◊ i ◊ a błędy kasujemy klawiszem *DELETE*;
— wiemy, że mikrokomputer *Spectrum* wykonuje polecenie *PRINT* i domyślamy się, że może być programowany w języku *Basic*.

Pytania i zadania

1. Jakie znaczenie ma kursor *K*, a jakie kursor *L*?
2. Jak przejść do trybu rozszerzonego (kursor *E*)?
3. Sprawdź, czy argumentem polecenia *PRINT* może być wyrażenie arytmetyczne. Czy do oddzielania części całkowitej liczby od części ułamkowej używamy kropki czy przecinka?
4. Sprawdź, czy argumentem polecenia *PRINT* może być tekst?
5. Sprawdź, czy działanie separatorów , (przecinek) i ; (średnik) oddzielających argumenty polecenia *PRINT* jest takie samo jak w mikrokomputerze *Meritum* (wykonaj polecenia podane w p. 1.1).
6. Spróbuj napisać i wykonać programy z p. 1.2.

1.5
Programujemy i piszemy przy użyciu kolorowych znaków

Mikrokomputer *Spectrum* programujemy, podobnie jak *Meritum*, w języku *Basic*. Między obydwoma implementacjami języka (na *Spectrum* i *Meritum*) istnieją niewielkie różnice, dotyczące głównie operacji na tekstach, które w *Spectrum* są zrealizowane trochę nietypowo, i operacji graficznych, co

wynika z innej organizacji wyświetlania w obydwu komputerach. W zakresie podstawowych instrukcji, obydwa mikrokomputery możemy programować tak samo. Na przykład bez zmian możemy uruchomić na *Spectrum* program konwersacji z p. 1.2

10 PRINT "JAK MASZ NA IMIE?"
20 INPUT X$
30 PRINT "HALO "; X$;",MILO MI CIE POZNAC"

Wykonanie programu rozpocznie się po napisaniu takiego samego polecenia *RUN*. Inny będzie tylko format konwersacji na ekranie i komunikat końcowy. Nieco inna jest też reakcja mikrokomputera na instrukcję *INPUT X$*. Mikrokomputer *Spectrum* rozpoznaje typ wprowadzanych argumentów i automatycznie wyświetla znaki cudzysłowu (" "), między którymi wpisuje się tekst podstawiany jako wartość zmiennej tekstowej *X$*.

W czasie pisania programu kolejne instrukcje są wyświetlane w górnej części ekranu. Ostatnia wprowadzona linia jest zaznaczona kursorem linii > wyświetlanym w lewej kolumnie tekstu. Położenie kursora możemy zmieniać, przesuwając go w górę lub w dół za pomocą klawiszy oznaczonych ⌃ i ⌄. Jest to ważne, gdyż linia wskazywana przez kursor > może być łatwo modyfikowana. Naciskając klawisz *EDIT* przenosimy tę linię do dolnej części ekranu, gdzie możemy ją poprawiać używając klawiszy ◊, ◊ i *DELETE*, tak samo jak to robiliśmy w poprzednim podrozdziale. Po zakończeniu wprowadzania poprawek naciskamy klawisz *ENTER* i linia wraca na swoje miejsce w programie. Kursor linii jest wyświetlany również po wyświetleniu programu poleceniem *LIST*. W niektórych sytuacjach kursor linii może zniknąć z ekranu. Możemy go z powrotem wywołać naciskając jeden z klawiszy ⌃ lub ⌄.

Spośród dotychczas poznanych instrukcji i poleceń: *PRINT*, *INPUT*, *GOTO*, *NEW*, *CLEAR*, *CLS*, *LIST*, wszystkie mogą być używane jako polecenia lub jako instrukcje programu przeznaczonego do wykonania przez mikrokomputer *Spectrum*. Jedynie w poleceniu *CLEAR* nie można, ale i nie trzeba rezerwować miejsca na zmienne tekstowe. Odpowiedni obszar pamięci jest rezerwowany automatycznie. Polecenie *CLEAR* z dodatkowym argumentem ma tu inne znaczenie, które poznamy w rozdz. 4.

Podobnie jak mikrokomputer *Meritum*, *Spectrum* jest przystosowany do współpracy z dowolnym magnetofonem. Podstawowe polecenia związane z obsługą magnetofonu są podobne

SAVE "nazwa"

powoduje zapisanie programu wraz z jego nazwą na taśmie,

LOAD "nazwa"

powoduje odszukanie programu i ładowanie do pamięci operacyjnej. Nazwy programów mogą się składać co najwyżej z 10 znaków, przy czym, w odróżnieniu od *Meritum*, wszystkie znaki są rozpoznawane. Wariantem polecenia *SAVE* jest polecenie

SAVE "nazwa" *LINE* numer

które również powoduje zapisanie programu na taśmie z tą różnicą, że wyko-

nane następnie polecenie *LOAD* nie tylko załaduje program, ale również automatycznie rozpocznie jego wykonanie od linii o podanym numerze. Do sprawdzenia poprawności zapisania programu na taśmie służy polecenie

VERIFY "nazwa"

które odczytuje program zapisany na taśmie i porównuje z programem znajdującym się aktualnie w pamięci operacyjnej.

Wykonanie polecenia *LOAD* powoduje ładowanie programu do pamięci i całkowite usunięcie programu zapisanego tam poprzednio. Polecenie

MERGE "nazwa"

również ładuje program z taśmy nie usuwając jednak z pamięci poprzedniego programu. Zmianie ulegają tylko te linie programu, których numery pokrywają się z numerami linii w programie ładowanym (wtedy, oczywiście, do pamięci jest wpisywana nowa linia z taśmy).

Wszystkie wymienione polecenia (łącznie z *SAVE*) kończymy naciśnięciem klawisza *ENTER*, po czym dopiero uruchamiamy magnetofon z odpowiednio przewiniętą taśmą. W przypadku polecenia *SAVE* mikrokomputer wyświetli na ekranie komunikat

Start tape, then press any key

co znaczy: *uruchom magnetofon, po czym naciśnij dowolny klawisz.*

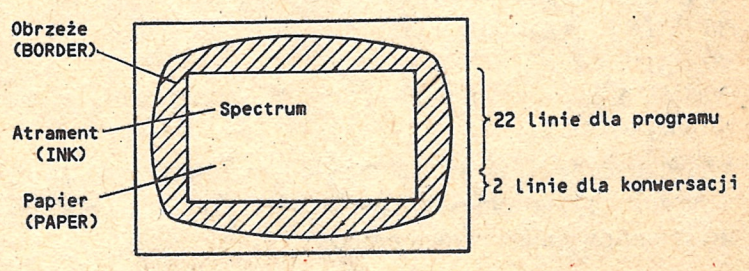
Operacja zapisu programu zostanie rozpoczęta dopiero po naciśnięciu dowolnego klawisza. Poprawne wykonanie operacji (każdej z czterech wymienionych) jest kwitowane komunikatem

0 OK, 0:1

Komunikat nie jest wyświetlany w przypadku automatycznego uruchamiania programu. W czasie ładowania (lub weryfikacji) mikrokomputer wyświetla u góry ekranu nazwy wszystkich napotkanych na taśmie programów lub zbiorów danych, które omija poszukując właściwego programu. Może to być bardzo użyteczne dla ustalenia zawartości nieznanej taśmy z programami.

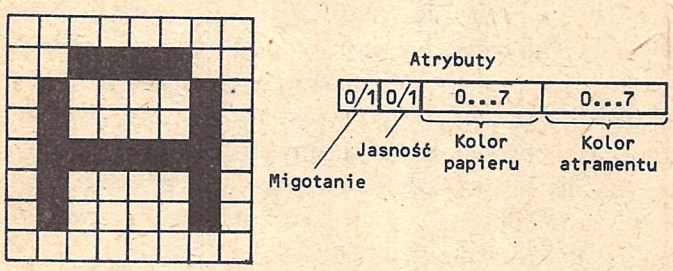
Ładowaniu programu z taśmy towarzyszą efekty wizualne (głównie kolorowe pasy) na ekranie telewizora. Przy okazji możemy więc zapoznać się ze strukturą ekranu i sposobem „kolorowania" obrazu.

Powierzchnia ekranu jest podzielona na dwa rozłączne pola nazwane **obrzeżem** (ang. *border*) i **papierem** (ang. *paper*). Obrzeże tworzy ramkę dookoła środkowego, użytecznego pola papieru (rys. 1.4), na którym możemy pisać znaki w **kolorze atramentu** (ang. *ink*). Początkowo, po włączeniu zasilania zarówno obrzeże jak i papier są wyświetlane w kolorze białym, natomiast



Rys. 1.4. Podział ekranu mikrokomputera *Spectrum*

atrament jest koloru czarnego. Przyporządkowanie kolorów może być zmienione za pomocą poleceń: *BORDER*, *PAPER* i *INK*. Argumentem każdego z tych poleceń jest liczba określająca kolor zgodnie z konwencją:
0 — czarny
1 — niebieski
2 — czerwony
3 — purpurowy
4 — zielony
5 — jasnoniebieski
6 — żółty
7 — biały
Liczby przypisane poszczególnym kolorom są tak dobrane, że na ekranie telewizora czarno-białego rosnącym numerom odpowiadają coraz jaśniejsze barwy. Na przykład, napisanie polecenia
BORDER 0
zmieni kolor obrzeża na czarny, czyniąc widoczną ramkę dookoła ekranu. Powierzchnia papieru jest podzielona na 24 linie po 32 znaki w linii. Jeżeli przyjrzymy się z bliska literkom wyświetlanym na ekranie, to zauważymy, że są one zbudowane z kropek. Każdy znak zajmuje kwadrat o rozmiarach 8 na 8, w którym odpowiedni wzór zostaje ułożony z kropek w kolorze atramentu. Pozostałe kropki mają kolor papieru (rys. 1.5). Polecenia *PAPER* i *INK* zmieniają kolor papieru i atramentu nie na całym ekranie, tylko w polach tych znaków, które zostaną napisane po wykonaniu poleceń *PAPER* i *INK*.



Rys. 1.5. Wzorzec litery i atrybuty

Zmiana nie dotyczy kolorów dwóch najniższych linii ekranu (polecenia i komunikaty komputera), w których kolor papieru jest taki sam jak kolor obrzeża (ramki). Na przykład, wykonanie programu
10 PAPER 0
20 INK 4
30 PRINT "ALA"
40 PAPER 2
50 INK 6
60 PRINT "EWA"
wyświetli imię *ALA* napisane zielonym kolorem na czarnym tle i *EWA* napisane żółtym kolorem na czerwonym tle. Powrót do białego papieru i czarnego atramentu wymaga dwóch poleceń

PAPER 7
INK 0
które nie zmienią oczywiście kolorów już wyświetlonych napisów *ALA* i *EWA*. Ciekawa i użyteczna (jeżeli dysponujemy telewizorem kolorowym) jest również możliwość umieszczania poleceń *PAPER* i *INK* wewnątrz instrukcji *PRINT*. Tak umieszczone polecenia są realizowane przez mikrokomputer tylko w trakcie wykonywania instrukcji *PRINT*. Nie mają natomiast żadnego wpływu na dalsze instrukcje programu. Na przykład program

```
10 PAPER 7
20 INK 0
30 PRINT "ALA"
40 PRINT INK 1; "IZA"; INK 5; "BELLA"
50 PRINT "EWA"
```

wyświetli na ekranie imię *IZABELLA* w kolorze częściowo ciemnoniebieskim (*IZA*), a częściowo jasnoniebieskim (*BELLA*). Imiona *ALA* i *EWA* zostaną natomiast wyświetlone w kolorze czarnym.

Oprócz koloru papieru i atramentu znak może wyróżniać się podwyższoną jasnością świecenia lub migotaniem. Polecenie

BRIGHT 1

powoduje wyświetlanie pisanych po nim znaków z większą jasnością.
Polecenie

FLASH 1

powoduje wyświetlanie znaków migających. Na przykład program

```
10 BRIGHT 0
20 FLASH 0
30 PRINT "A"; BRIGHT 1; "N"; FLASH 1; "N"; BRIGHT 0; "A"
```

spowoduje wyświetlenie imienia *ANNA*, w którym pierwsze *A* jest wyświetlone normalnie, pierwsze *N* z podwyższoną jasnością świecenia, drugie *N* miga (z podwyższoną jasnością świecenia), a drugie *A* miga z normalną jasnością świecenia.

Napisanie nowego znaku w miejscu, w którym już znajdował się jakiś znak, powoduje wymazanie „starego" znaku i wpisanie w jego miejsce wzorca kropek nowego.

Jedynym odstępstwem od tej zasady jest działanie instrukcji *PRINT* po wykonaniu polecenia

OVER 1

Pisane teraz znaki nie wymazują znaków poprzednio istniejących w tym miejscu, lecz w pewien sposób dodają się do nich. Reguły tego specyficznego dodawania są następujące:

— różne kolory kropki w tym samym punkcie (tzn. kropka w kolorze papieru w starym znaku i kropka w kolorze atramentu w nowym lub odwrotnie) dają w rezultacie kropkę w kolorze atramentu,
— takie same kolory kropek (papier-papier lub atrament-atrament) dają w rezultacie kropkę w kolorze papieru.

Ostatnią funkcją zmieniającą sposób wyświetlania jest funkcja *INVERSE*. Polecenie
INVERSE 1
powoduje, że od tej chwili zamiast normalnych znaków będą wyświetlane ich negatywy. Kropki atramentowe zostaną postawione tam, gdzie w normalnym znaku były kropki w kolorze papieru i odwrotnie. Efekt polecenia *INVERSE 1* utrzymuje się aż do chwili wykonania polecenia *INVERSE 0*. Polecenie *INVERSE 1* może być używane wewnątrz instrukcji *PRINT* analogicznie jak poprzednie polecenia.

Podsumowanie

— polecenia *SAVE* i *VERIFY* pozwalają zapisać program na taśmę i sprawdzić jego poprawność;
— polecenia *LOAD* i *MERGE* ładują program do pamięci mikrokomputera;
— polecenia: *BORDER*, *PAPER*, *INK* określają kolor ramki (obrzeża ekranu), papieru i atramentu;
— polecenia: *BRIGHT*, *FLASH* określają jasność i migotanie znaków;
— polecenia: *OVER*, *INVERSE* zmieniają sposób wpisywania kropek wzorca znaku;
— wszystkie polecenia dotyczące koloru i sposobu wyświetlania znaków mogą być też używane w programie jako instrukcje lub funkcje wewnątrz instrukcji *PRINT*.

Pytania i zadania

1. Napisz program, który spowoduje wyświetlenie na ekranie czterech białych gwiazdek na czarnym tle i czterech czarnych gwiazdek na białym tle.
2. Zapisz program z zad. 1 na taśmie magnetofonowej, po czym sprawdź poprawność zapisu.
3. Napisz program, który spowoduje odczytanie liczby (z zakresu 0...7) i ustalenie koloru ramki, zgodnie z wczytaną wartością (0 — czarny, ..., 7 — biały).
4. Napisz program, który wyświetli na środku wiersza migające słowo *ALARM* (białe litery na czarnym tle) podkreślone pięcioma czarnymi gwiazdkami na białym tle.
5. Napisz program z zad. 1 i zapisz go na taśmie pod nazwą *G4*; napisz program z zad. 4 numerując linie: *100, 110* i zapisz go na taśmę pod nazwą *AG*. Załaduj program *G4* poleceniem *LOAD* i wykonaj. Załaduj program *AG* poleceniem *MERGE* i wykonaj. Załaduj program *AG* poleceniem *LOAD* i wykonaj. Wyjaśnij różnicę w działaniu poleceń *LOAD* i *MERGE*.

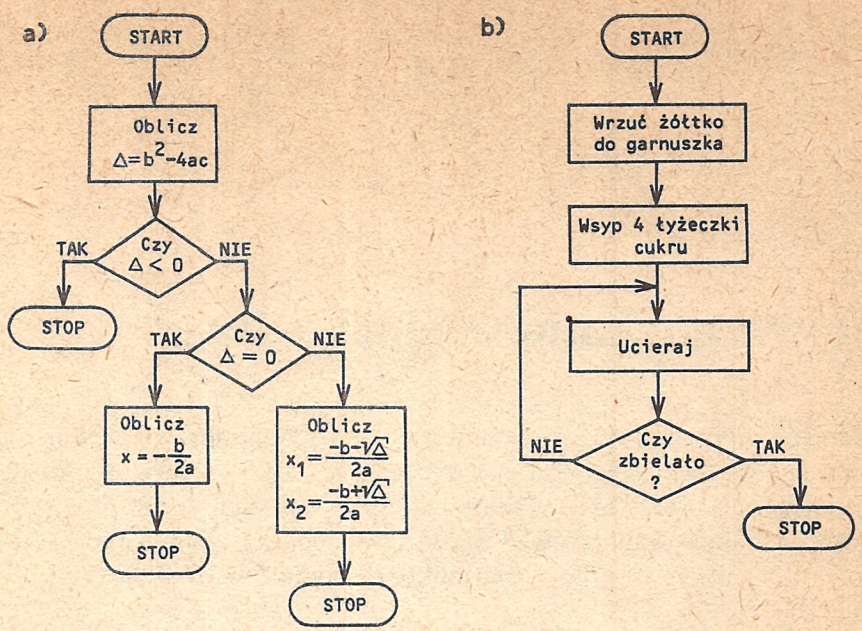
2
Programowanie w języku Basic

Poznamy teraz sposób programowania mikrokomputerów *Meritum* i *Spectrum* w języku *Basic*. Materiał zawarty w p. 2.1...2.5 dotyczy obydwóch mikrokomputerów, może być też w większości wykorzystany przy programowaniu innych mikrokomputerów. Ostatnie dwa punkty poświęcono specyficznym cechom instrukcji graficznych mikrokomputerów *Meritum* (2.6) i *Spectrum* (2.7).

2.1
Zanim zaczniemy programować

Każdy program, niezależnie od języka programowania, jest drobiazgowym i zawierającym wszystkie szczegóły opisem metody rozwiązania problemu przy użyciu operacji wykonywanych przez komputer. Jest oczywiste, że aby taki szczegółowy przepis postępowania (nazywany **algorytmem**) podać, musimy najpierw sami dokładnie wiedzieć, co chcemy osiągnąć i jaką metodą do tego celu dojść. Poza bardzo prostymi przypadkami zanim przystąpimy do napisania programu musimy bardzo starannie sformułować problem, który staramy się rozwiązać, a następnie włożyć wiele wysiłku w opracowanie metody jego rozwiązania. Dopiero potem możemy przystąpić do programowania.

Bardzo pomocnym narzędziem na etapie poprzedzającym programowanie są tzw. sieci działań. **Sieci działań** stanowią graficzny sposób zapisywania algorytmów. Na rysunku używamy kilku rodzajów symboli: prostokątów do zapisywania czynności, rombów do sprawdzania warunków i strzałek do wskazania następstwa akcji. Za pomocą sieci działań możemy opisywać dowolne algorytmy, np. rozwiązywania równań kwadratowych (rys. 2.1a) lub przyrządzania kogla-mogla (rys. 2.1*b*). Po napisaniu programu realizującego algorytm opisany wstępnie siecią działań, sieć staje się automatycznie elementem dokumentacji programu. W przyszłości pozwoli ona łatwo przypomnieć sobie, co dany program robi i jak go zmienić: rozszerzyć lub ulepszyć.



Rys. 2.1. Przykłady algorytmów: *a)* rozwiązywania równania kwadratowego $ax^2 + bx + c = 0$; *b)* przyrządzania kogla-mogla

Napisanie programu nie jest jeszcze końcowym etapem pracy. Bardzo rzadko zdarza się, że świeżo napisany program od razu działa. Z reguły roi się on od błędów i dopiero długotrwały proces **uruchamiania programu** doprowadza nasze dzieło do stanu używalności. Problemem związanym z uruchamianiem programu jest sprawa weryfikacji i interpretacji otrzymanych wyników. Przypuśćmy, że napisaliśmy jakiś program i doprowadziliśmy go do stanu, w którym komputer wyświetlił jakiś wynik. Ale czy jest to wynik poprawny? Odpowiedź na to pytanie nie jest na ogół prosta. Zanim nabierzemy do programu zaufania, musimy go najpierw przetestować, tzn. kilkakrotnie wykonać wprowadzając do niego takie dane początkowe, dla których znamy rozwiązanie naszego problemu. Jeżeli za każdym razem odpowiedź komputera będzie zgodna z oczekiwaniami, to możemy z dużą dozą prawdopodobieństwa przyjąć, że poprawnie rozwiązuje on nasz problem.

Aby zilustrować trudności w opracowaniu poprawnego algorytmu wróćmy do przykładu z rys. 2.1*a*. Przedstawiony tam algorytm rozwiązywania równań kwadratowych jest powszechnie znany i nie budzi sprzeciwu. Czy jednak jest w pełni poprawny? Sprawdźmy. Dla danych 1,3,2 stanowiących współczynniki równania

$$x^2 + 3x + 2 = 0$$

otrzymamy poprawny wynik $x_1 = -2, x_2 = -1$. Dla danych 1,1,1 opisujących równanie

$$x^2 + x + 1 = 0$$

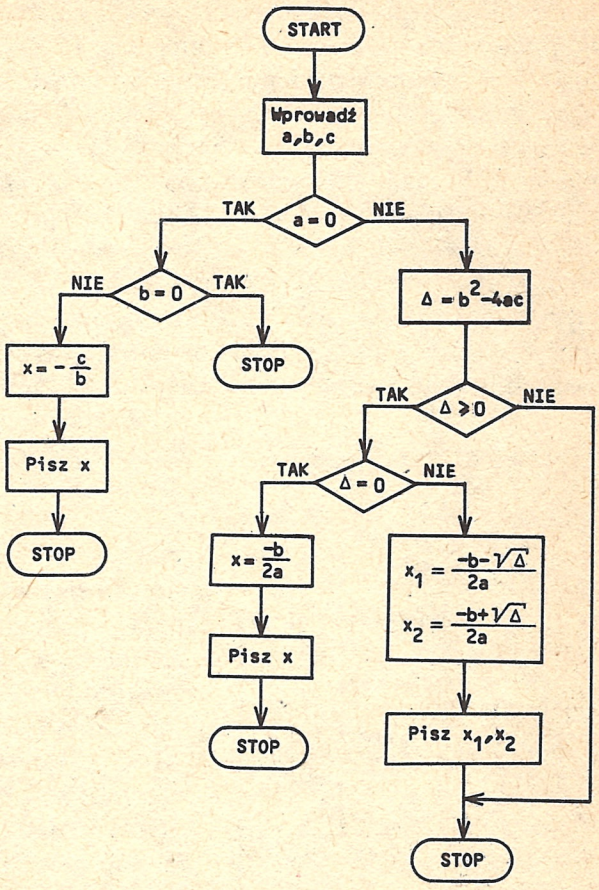
nie otrzymamy co prawda wyniku, ale algorytm osiągnie blok *STOP*, co odczytamy jako brak (nieistnienie) pierwiastków. Gorzej będzie przy danych 0,3,1. Wyróżnik $\Delta = 9$ jest dodatni i idąc za strzałkami sieci działań dochodzimy do operacji

$$x_1 = \frac{-b - \sqrt{\Delta}}{2a} \qquad x_2 = \frac{-b + \sqrt{\Delta}}{2a}$$

W mianowniku mamy zero, dzielenie jest niewykonalne i algorytm okazuje się błędny. Przyczyna jest prosta. Dane 0,3,1 odpowiadają równaniu

$$3x + 1 = 0$$

które nie jest w ogóle równaniem kwadratowym, lecz liniowym. **W pełni poprawny algorytm powinien zawierać wstępne sprawdzenie danych i gwarantować osiągnięcie końca (bloku *STOP*) dla każdego zestawu danych początkowych.** W naszym przykładzie możemy łatwo dołączyć do algorytmu operacje prowadzące do poprawnego rozwiązania równania liniowego (rys.2.2).



Rys. 2.2. Algorytm rozwiązywania równań stopnia nie wyższego niż drugi

Przy okazji przykładu z równaniami zwróćmy uwagę na charakterystyczną cechę algorytmu (a w perspektywie programu). Równanie kwadratowe, które normalnie zapisujemy

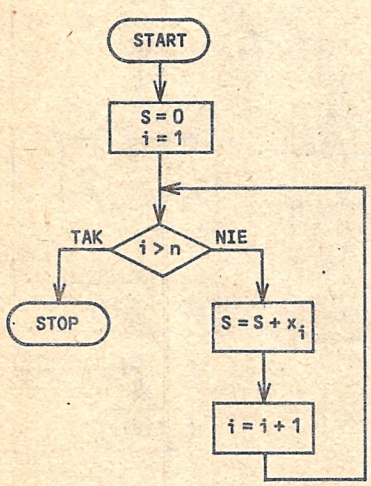
$$ax^2 + bx + c = 0$$

jest tu reprezentowane wyłącznie przez trzy liczby a, b, c z pominięciem czynników x^2 i x. Jest to zgodne z ustaleniami matematyki: równanie jest w pełni określone przez wartości współczynników, a x jest tzw. zmienną związaną, którą piszemy tylko dla zwiększenia przejrzystości zapisu.

Oprócz zasadniczych trudności z poprawnością algorytmów, w początkowym okresie programowania będziemy mieli spore trudności z przystosowaniem naszego sposobu myślenia do ściśle sekwencyjnego sposobu zapisywania algorytmów, a później programów dla komputera. Rozważmy prosty przykład zadania zsumowania wartości n liczb $x_1, x_2,\ldots,x_n$. W notacji matematycznej piszemy

$$S = x_1 + x_2 + \ldots + x_n = \sum_{i=1}^{n} x_i$$

i rozumiemy, że dodawanie zostanie wykonane w dowolny sposób, np. jednocześnie dodamy do siebie wartości n liczb. Przy opracowaniu algorytmu dla komputera nie możemy tak rozumować. Program będzie tworzony jako ciąg instrukcji wykonywanych nie jednocześnie, ale kolejno, jedna po drugiej. Na nasze zadanie musimy więc spojrzeć nie jak na jedno dodawanie n liczb, lecz jak na n dodawań kolejnych liczb x_i dla $i = 1,\ldots,n$. To znaczy najpierw do pierwszej liczby dodamy drugą, następnie do otrzymanej w ten sposób sumy dodamy trzecią, następnie do sumy trzech pierwszych liczb dodamy czwartą liczbę itd. Po każdym kolejnym kroku musimy pamiętać tylko jedną wartość: sumę wszystkich dodanych do tej chwili liczb. Początkowo, przed rozpoczęciem dodawania wartość tej sumy wynosi zero. Opisany słownie algorytm możemy przedstawić graficznie w postaci sieci działań (rys.2.3). Suma dodawanych liczb jest gromadzona jako wartość zmiennej S.



Rys. 2.3. Algorytm sumowania ciągu liczb

Zwróćmy uwagę na specyficzny zapis dwóch operacji w naszej sieci

$$S = S + x_i$$
$$i = i + 1$$

W każdej z nich po obu stronach znaku = (równa się) występuje ta sama zmienna. W notacji matematycznej taki zapis oznacza równanie, w językach programowania jego sens jest inny. Jest to **polecenie podstawienia** wartości wyrażenia arytmetycznego stojącego po prawej stronie znaku = (który będziemy nazywać **znakiem podstawienia**), jako nowej wartości zmiennej wymienionej po stronie lewej. W naszym przykładzie: za S podstaw S plus kolejna liczba x_i. Wykonanie polecenia podstawienia przebiega w dwóch wyraźnie rozdzielonych i następujących po sobie krokach:

— najpierw jest obliczana wartość wyrażenia arytmetycznego,
— potem następuje zmiana wartości zmiennej.

Znak = (podstawienie) nie oznacza tu więc relacji równości, lecz jest znakiem operacji zmiany wartości zmiennej. W ten sposób należy interpretować wszystkie wystąpienia znaku = (podstawienie) w prostokątnych blokach wykonaw-

czych sieci działań. Jedynie w blokach warunkowych (romby) znak = zachowuje tradycyjne znaczenie znaku relacji równości.

Konstrukcja sieci działań z rys. 2.3 jest istotnie różna od konstrukcji poprzedniej sieci: strzałka wychodząca z bloku zwiększania wartości wskaźnika ($i = i + 1$) zawraca nas do bloków, które już były wykonane. Dzięki temu ten sam blok dodawania kolejnej liczby jest wykonywany (obiegany) wielokrotnie, dodając za każdym razem inną liczbę x_i. **Wielokrotnie wykonywany fragment algorytmu (programu) nosi nazwę pętli lub iteracji.** Pętla jest jedną z fundamentalnych koncepcji programowania i spotykamy ją w niemal każdym większym programie. Przy projektowaniu pętli **musimy zawsze zagwarantować jej zakończenie.** W naszym programie ten warunek jest spełniony: w kolejnych obiegach pętli zwiększamy wartość wskaźnika i o 1 poczynając od wartości 0 tak, że po skończonej liczbie obiegów wartość ta przekroczy wartość liczby n, która jest z założenia dodatnia. Algorytm nie gwarantujący zakończenia pętli może nigdy nie osiągnąć końca i jest oczywiście błędny. Umieszczenie bloku badania warunku zakończenia sumowania ($i>n$) przed pierwszym dodawaniem gwarantuje poprawne działanie algorytmu również w przypadku, gdy $n = 0$ i nie ma czego sumować.

Podsumowanie

— poznaliśmy sposób zapisywania algorytmów za pomocą sieci działań;
— wiemy, że poprawnie skonstruowany algorytm powinien gwarantować osiągnięcie bloku *STOP*;
— poznaliśmy konstrukcję pętli;
— wiemy, że znak = ma w językach programowania podwójne znaczenie: może by znakiem relacji równości albo znakiem podstawienia.

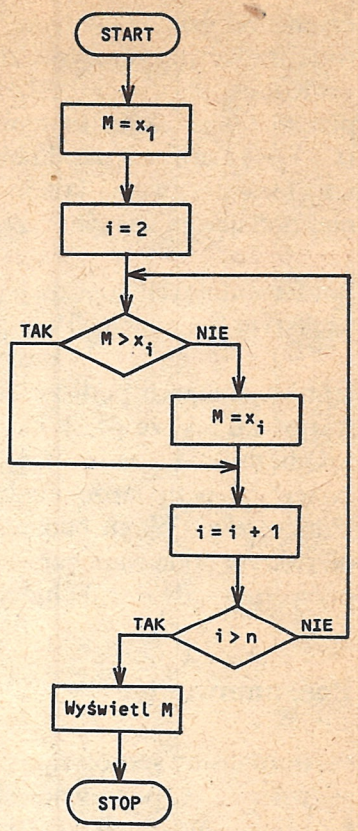
Pytania i zadania

1. Z jakich elementów budujemy sieci działań?
2. Narysuj sieć działań rozwiązującą układ równań:
   $a_1x + b_1y = c_1$; $a_2x + b_2y = c_2$
   (wskazówka: najpierw przypomnij sobie jak rozwiązujemy równania, a dopiero potem zacznij rysować sieć działań).
3. W sieci działań z rys. 2.3 nigdzie nie nadajemy wartości zmiennej n. Dopisz bezpośrednio za blokiem *START* nowy blok, zawierający operację Wprowadź n. Dla jakiego zakresu wartości zmiennej n algorytm jest poprawny?
4. Wartość liczby dwójkowej $b_n...b_1b_0$ obliczamy jako

$$w = b_0 + 2b_1 + \ldots + 2^n b_n = \sum_{i=0}^{n} 2^i b_i$$

Narysuj sieć działań algorytmu obliczającego wartość liczby dwójkowej $b_n \ldots b_1 b_0$.

5. Funkcja silnia jest zdefiniowana dla liczb naturalnych jako:
   $n! = n(n-1)(n-2)\ldots 2 \cdot 1$ i $0! = 1$
   np. $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$. Narysuj sieć działań algorytmu obliczającego wartość tej funkcji dla argumentu n.
6. Jaki algorytm realizuje sieć działań przedstawiona na rys. 2.4?



Rys. 2.4. Rysunek do zadania 6

2.2
Elementy języka

Wszystkie instrukcje programu wykonują pewne działania na argumentach wymienionych w treści instrukcji. **Argumenty** te mogą się składać z następujących elementów:
— stałe,
— zmienne,
— wyrażenia arytmetyczne,
— funkcje standardowe.
Sposób korzystania z tych elementów w programie jest ściśle określony przez składnię instrukcji. Równie ściśle jest określony sposób zapisywania tych elementów w instrukcjach.

Stałe występujące w języku *Basic* możemy podzielić na liczby i teksty. Liczby używane w obliczeniach są liczbami rzeczywistymi, zapisywanymi w sy-

stemie dziesiętnym. Zgodnie ze zwyczajem przyjętym w języku angielskim do oddzielenia części całkowitej od ułamkowej używa się kropki. Jeżeli jest to potrzebne, możemy też użyć zapisu wykładniczego, typu: $2.15 \cdot 10^{-2}$ z tym, że zamiast 10 musimy użyć litery E. Tak więc wartość liczby π możemy zapisać

3.14 lub 0.314 E1

a wartość liczby Avogadro

6.022169E+23

Liczby ujemne wyróżniamy jak zwykle znakiem − (minus).

Stałe tekstowe zapisujemy jako dowolny ciąg liter, cyfr i innych znaków pisarskich umieszczonych w cudzysłowie, np.

"IZA" *"1025"* *"ALA"*

Zwróćmy uwagę, że zapis *"1025"* nie oznacza liczby i próba wykonania na nim działań arytmetycznych, np. w poleceniu

PRINT "1025" −5

nie ma sensu i spowoduje zasygnalizowanie błędu.

Stałe mogą być w programie podstawiane jako wartości zmiennych. Wszystkie **zmienne** są identyfikowane przez indywidualne nazwy, przy czym nazwą może być pojedyncza litera alfabetu łacińskiego:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

dwie litery albo litera i cyfra, np.

A *AX* *A2*

Podobnie jak stałe, tak i zmienne dzielimy na liczbowe (o wartościach liczbowych) — nazywane też zmiennymi prostymi — i tekstowe. Zmienne obydwu typów są elementami istotnie różnymi. W zapisie języka *Basic* nazwy zmiennych tekstowych muszą być zakończone znakiem *$* (sybol dolara), np.

A$* *Z$ *X$*

Na *Spectrum* nazwy zmienych prostych mogą być dłuższe niż dwa znaki, za to nazwy zmiennych tekstowych są ograniczone do jednej litery z dodanym symbolem dolara.

Zarówno stałe jak i zmienne proste mogą być używane do budowy **wyrażeń arytmetycznych**. Reguły budowy wyrażeń są zgodne ze standardowymi regułami arytmetyki, z następującymi ograniczeniami:

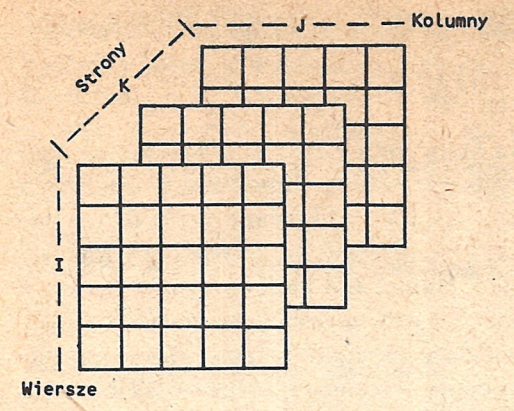
— do oznaczania działań używamy operatorów:
+ (dodawanie), − (odejmowanie), * (mnożenie), / (dzielenie), ↑ (potęgowanie);

— nie wolno opuszczać symbolu mnożenia, np. wyrażenie $AX + B$ trzeba zapisać $A*X + B$;

— wolno używać tylko nawiasów okrągłych;

— dwa operatory nie mogą występować obok siebie, np. ułamek $\frac{A-B}{-2}$ trzeba zapisać $(A-B) / (-2)$.

Reguły obliczania wyrażeń arytmetycznych są również zgodne z regułami znanymi z matematyki z tym uzupełnieniem, że jeżeli rodzaj operatorów nie określa jednoznacznie kolejności działań, to działania są wykonywane w ko-



Rys. 2.7. Tablica trójwymiarowa

ny, tak że każdy element tablicy jest jednoznacznie określony trzema indeksami *A (I, J, K)*. Wartości indeksów tablic rozpoczynają się od 0 w wersji języka *Basic* dla *Meritum* i od 1 w wersji dla *Spectrum*. W praktyce tablice o liczbie indeksów większej niż 2 są używane bardzo rzadko. Elementy tablic, nazywane **zmiennymi indeksowanymi**, mogą być używane w wyrażeniach arytmetycznych tak samo jak zmienne proste, np. w programie

10 INPUT A(1), A(2)
20 PRINT A(1) + A(2), A(1)↑3 + A(2)↑3, (A(1) + A(2))↑3

użyte są wyrażenia arytmetyczne, które w tradycyjnej notacji matematycznej zapiszemy

$$a_1 + a_2 \qquad a_1^3 + a_2^3 \qquad (a_1 + a_2)^3$$

Na zakończenie rozważymy ważny w praktyce problem zakresu i dokładności obliczeń. Sposób pamiętania liczb wewnątrz komputera jest zbliżony do zapisu w postaci wykładniczej (typu $2{,}15 \cdot 10^2$), przy stałej liczbie cyfr przewidzianych na napisanie części całkowitej, ułamkowej i wykładnika (w rzeczywistości jest to zapis dwójkowy, co jednak w niczym nie zmienia sytuacji). Konsekwencją takiej metody zapisu jest ograniczenie zakresu liczb. W większości mikrokomputerów wszystkie wartości liczbowe muszą zawierać się w zakresie od -10^{38} do $+10^{38}$. Dalszą konsekwencją jest brak ułamków wymiernych, takich jak np. $\frac{1}{3}$, $\frac{2}{7}$, itp. i nieuchronne pojawienie się błędów w czasie obliczeń.

Na przykład w *Meritum* $\frac{1}{3} + \frac{1}{3}$ równa się co prawda $\frac{2}{3}$, ale już $2 + \frac{1}{3} + \frac{1}{3}$ nie równa się $2 + \frac{2}{3}$. Polecenie

PRINT (2 + 1/3 + 1/3) − (2 + 2/3)

spowoduje wyświetlenie wyniku

−2.38419E−07

Analogiczne polecenie dla mikrokomputera *Spectrum* spowoduje obliczenie wartości −9.3132257E−10.
W ramach eksperymentu spróbujmy też napisać

PRINT 1E12 + 1

co odpowiada działaniu $10^{12} + 1$. Na ekranie otrzymamy jednak wynik *1E + 12*, gdyż dodana jedynka nie mieści się w tej liczbie cyfr, które są używane do zapisu liczby.

Należy podkreślić, że obydwa wymienione ograniczenia (zakres i dokładność) nie są związane z żadnymi cechami języka *Basic*, lecz wynikają ze sposobu pamiętania liczb przez komputer. Problem ten wystąpi więc przy programowaniu komputerów w dowolnym języku.

Podsumowanie

— najprostszymi elementami języka *Basic* są: stałe, zmienne proste i zmienne tekstowe;
— bardziej złożoną budowę mają zmienne indeksowane, stanowiące elementy tablic;
— stałe, zmienne proste i zmienne indeksowane są elementami, z których budujemy wyrażenia arytmetyczne;
— wyrażenia arytmetyczne budujemy zgodnie z regułami arytmetyki, używając operatorów działań: +, −, ∗, /, ↑, funkcji standardowych i nawiasów okrągłych;
— zarówno zakres jak i dokładność liczb są ograniczone.

Pytania i zadania

1. Dlaczego wszystkie wymienione zapisy stałych są błędne?

   1,2 2^2 2.2E1.2 + −2.2 1/2

2. Zapisz w notacji wykładniczej z wykładnikiem równym 2:

   125 314.15 34E1 −34E3 12845E−3

3. Dlaczego wszystkie podane nazwy zmiennych są błędne?

   1A Δ A/3 oś A_2 A$3 A(−1)

4. Które z podanych wyrażeń są poprawne, a które błędne?

   (A+2/3) $X\uparrow 2 - 3A\uparrow 5$ $B\uparrow 2 - 5 * A * C$ $\frac{1}{2} + 3 * B$

   $(A + B)\,(C + X)$ $SIN\,(X) - COS\,(\ )$ $(A/-2) * 3$

   $(1/X\uparrow 2 - (3 * X\uparrow 2 + 5) / (6 * 8)\uparrow 2$ $(3 + (A + (5 + SQR\,(X))))$

5. Sprawdź poprawność wyrażeń z pytania 4 za pomocą programu

   10 INPUT A,B,C,X
   20 PRINT wyrażenie

6. Sprawdź, jak działa operator + na stałych i zmiennych tekstowych, np.

   PRINT "IZA" + "BELLA", PRINT "10" + "25" + "R"

2.3
Podstawowe instrukcje

W punkcie 2.1 przekonaliśmy się, że dogodnym narzędziem przedstawiania algorytmów obliczeniowych są sieci działań. Algorytmy wykonywane przez komputer muszą być jednak zapisane w postaci programów. Aby umożliwić przedstawienie dowolnych algorytmów, zestaw instrukcji, z których budujemy programy musi umożliwiać opisanie wszystkich elementów sieci działań.

Zasadniczą czynnością wykonywaną w bloku wykonawczym sieci działań (rysowanym w postaci prostokąta) jest zmiana wartości zmiennych. W języku *Basic* czynność tę wykonuje **instrukcja podstawienia *LET*** o postaci

LET zm = wa

gdzie *zm* oznacza nazwę dowolnej zmiennej prostej lub indeksowanej, a *wa* dowolne wyrażenie arytmetyczne. Słowo *LET* znaczy *NIECH* (*NIECH zm* stanie się równe *wa*). Szczególnym przypadkiem wyrażenia arytmetycznego może być stała lub inna zmienna, np.

```
10 LET X = 4
20 LET Y = SQR(X) + 6
30 LET X = X/Y
```

Znak = użyty w treści instrukcji ma znaczenie znaku podstawienia (por. p. 2.1). Po dopisaniu do poprzedniego programu instrukcji

```
40 PRINT X, Y, X + 1
```

otrzymamy na ekranie wynik

```
 0.5          8          1.5
```

W większości implementacji języka *Basic* słowo *LET* może zostać pominięte; jednym z nielicznych wyjątków jest mikrokomputer *Spectrum*, w którym użycie słowa *LET* jest obowiązkowe.

Podobną postać ma instrukcja podstawienia wartości tekstowej

LET zt = wt

gdzie *zt* oznacza nazwę dowolnej zmiennej tekstowej, a *wt* dowolne wyrażenie tekstowe, np.

```
10 LET A$ = "10" + "25"
20 LET B$ = A$ + "R"
```

W bloku warunkowym sieci działań (romb) następuje wybór dalszej drogi algorytmu, zależnie od spełnienia lub nie spełnienia pewnego warunku. Na przykład w pierwszym rombie na rys. 2.2 przyrównujemy wartość zmiennej *A* do zera i decydujemy: jeżeli $A = 0$, to idziemy drogą wskazaną przez strzałkę z napisem *TAK*, jeżeli $A \neq 0$, to idziemy drogą wskazaną przez strzałkę z napisem *NIE*. Odpowiednikiem bloku warunkowego jest w języku *Basic* **instrukcja warunkowa *IF*** o postaci

IF warunek THEN numer

powodująca sprawdzenie *warunku* podanego za słowem *IF* (*jeżeli*) i — jeżeli jest on spełniony — przejście do instrukcji o *numerze* podanym za słowem *THEN* (*to*). Jeżeli warunek nie jest spełniony, to następną wykonaną przez komputer instrukcją będzie instrukcja następna (wg kolejności numeracji) po instrucji *IF*, np.

```
10 IF X>0 THEN 30
20 LET X = -X
30 PRINT "MODUL X ="; X
```

Jeżeli $X>0$, to następną instrukcją, jaką wykona komputer, będzie instrukcja umieszczona w linii o numerze *30*, jeżeli natomiast $X\leq 0$, to instrukcja z linii *20*.

Warunki w instrukcji *IF* możemy budować używając następujących symboli relacji:

= równa się
< mniejsze
> większe
< > nie równa się
<= mniejsze lub równe
>= większe lub równe

Podstawowe symbole są zaczerpnięte z tradycyjnej notacji matematycznej. Pozostałe stanowią ich kombinacje.

Większość implementacji języka *Basic* pozwala również używać nieco inną postać instrukcji warunkowej o budowie

IF warunek THEN instrukcja

której wykonanie polega na sprawdzeniu *warunku* podanego po słowie *IF* i wykonaniu *instrukcji* podanej po słowie *THEN*, jeżeli warunek jest spełniony, lub niewykonaniu tej *instrukcji*, jeżeli warunek nie jest spełniony. Program obliczenia wartości bezwzględnej możemy teraz zapisać

```
10 IF X < 0 THEN LET X = -X
20 PRINT "MODUL X ="; X
```

Jako instrukcja uwarunkowana, podana po słowie *THEN*, może wystąpić dowolna instrukcja języka *Basic*.

Instrukcja warunkowa pozwala zmienić kolejność wykonywania instrukcji programu wyznaczoną przez ich numerację. Zmiana taka może nastąpi również w wyniku wykonania bezwarunkowej **instrukcji skoku *GOTO*** o budowie

GOTO numer

Następną po instrukcji *GOTO* (to znaczy *idź do*) będzie zawsze instrukcja o *numerze* podanym za zwrotem *GOTO*.

Zauważmy, że instrukcja warunkowa

IF warunek THEN GOTO numer

jest w pełni równoważna pierwszej omówionej postaci instrukcji *IF*:

IF warunek THEN numer

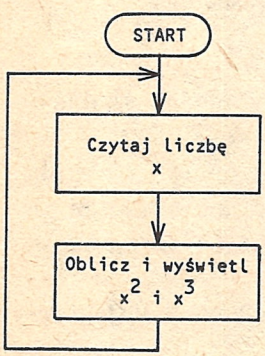
Jest to szczególnie ważne dla użytkowników mikrokomputera *Spectrum*, gdyż instrukcja *IF* w pierwszej postaci nie jest przez ten komputer wykonywana.

Podany zestaw instrukcji wystarczy nam do napisania programów realizujących sieci działań z rys. 2.2 i 2.3. Pierwszy z nich może mieć postać

```
10 INPUT A, B, C
20 IF A = 0 THEN GOTO 300
30 LET D = B↑2 - 4*A* C
40 IF D<0 THEN GOTO 100
50 IF D = 0 THEN GOTO 200
60 LET D = SQR (D)
70 LET X1 = (-B - D) / (2 * A)
80 LET X2 = (-B + D) / (2 * A)
90 PRINT X1, X2
100 STOP
200 LET X = (-B) / (2 * A)
210 PRINT X
220 STOP
300 IF B = 0 THEN STOP
310 LET X = (-C) / B
320 PRINT X
330 STOP
```

Program dokładnie odwzorowuje sieć działań z rys. 2.2 z jedną tylko różnicą: w linii 60 została wyodrębniona operacja obliczenia pierwiastka z delty (oznaczonej *D*).

Nową instrukcją, która pojawia się w programie, jest **instrukcja zakończenia programu *STOP***. To znaczy, komputer wyświetli odpowiedni komunikat i przejdzie w stan oczekiwania na nowe polecenia lub instrukcje. Zarówno zawartość pamięci programu jak i pamięci zmiennych zostaje zachowana. Ten sam efekt zatrzymania wykonania programu możemy osiągnąć naciskając klawisz *BREAK*. W naszym przykładzie jest to niepotrzebne, jak również niemożliwe, gdyż program wykona się tak szybko, że nie zdążymy w tym czasie nacisnąć żadnego klawisza. Gdybyśmy jednak napisali program obliczający kwadraty i sześciany liczb o postaci (rys. 2.8)



Rys. 2.8. Sieć działań programu obliczającego kwadraty i sześciany liczb

```
10 INPUT X
20 PRINT X↑2, X↑3
30 GOTO 10
```

to przerwanie wykonania przez naciśnięcie klawisza *BREAK* byłoby jedyną możliwością.* Program ten jest przykładem niekończącej się pętli, której w dobrze napisanych programach należy oczywiście unikać. W obydwu przypadkach zatrzymania wykonywania programu: czy to przez instrukcję *STOP*, czy

* **Uwaga:** w mikrokomputerze *Spectrum* klawisz *BREAK* nie może przerwać wykonania instrukcji *INPUT*. Przerwanie wykonania tego programu jest więc niemożliwe.

też przez naciśnięcie klawisza *BREAK*, możliwe jest jego wznowienie poleceniem

CONT

(co stanowi skrót od *Continue — kontynuuj*). Wznowienie wykonania programu następuje dokładnie od miejsca, w którym nastąpiło jego zatrzymanie.

Program realizujący sieć działań z rys. 2.3 jest bardzo krótki

```
100 LET S = 0
110 LET I = 1
120 IF I > N THEN STOP
130 LET S = S + X(I)
140 LET I = I + 1
150 GOTO 120
```

jednak nie może być w tej postaci uruchomiony, gdyż brakuje w nim instrukcji wprowadzania danych: liczb $x_1,...x_n$ oraz wartości n. Nie będziemy się tym problemem zajmować, gdyż zadanie sumowania ciągu liczb występuje na ogół jako część większego programu, w którym wartości te są obliczane przez inne fragmenty programu.

Z użyciem zmiennych indeksowanych jest związany dodatkowy problem. Z podanego w p. 2.2 określenia tablicy nie wynika, z ilu elementów się ona składa. Dla komputera jest to informacja bardzo istotna, gdyż określa rozmiar pola, jakie należy zarezerwować w pamięci zmiennych dla zapisania (zapamiętania) całej tablicy. Z tego powodu, z reguły wymaga się wstępnego zadeklarowania każdej używanej w programie tablicy. Służy do tego **deklaracja rozmiaru tablicy *DIM*** (od *Dimension — rozmiar*), o postaci

DIM lista tablic

gdzie *lista tablic* zawiera nazwy tablic wraz z maksymalnymi wartościami wszystkich indeksów, np.

```
10 DIM A(2,3), B(10)
20 DIM X(15)
```

Dla sprawdzenia funkcjonowania deklaracji dopiszmy np. instrukcję

```
30 LET X(16) = 10
```

Po uruchomieniu programu poleceniem *RUN* otrzymamy komunikat

3 Subscript wrong, 30:1 (*Spectrum*)
BS ERROR IN 30 (*Meritum*)

sygnalizujący błąd przekroczenia zakresu indeksu w linii 30 programu — zgodnie z deklaracją w linii 20 maksymalna wartość indeksu może wynosić tylko 15, element tablicy *X (16)* po prostu nie istnieje.

Powróćmy do przykładu programu sumowania liczb. Pierwsza instrukcja nadaje wartość początkową 0 zmiennej *S*, służącej do sumowania. Sumowanie jest realizowane przez instrukcję z numerem 130. Pozostałe cztery instrukcje są związane z organizacją wykonania pętli. Możemy tu wyróżnić:

— nadanie początkowej wartości indeksu (110);
— sprawdzenie przekroczenia wartości końcowej indeksu (120);
— zwiększenie wartości indeksu (140);

— powrót na początek pętli do operacji sprawdzenia przekroczenia wartości końcowej (150);

Konstrukcja pętli ze zwiększaną wartością indeksu jest tak często spotykana w praktyce, że w języku *Basic* istnieje para specjalnych **instrukcji *FOR-NEXT* otwierających i zamykających pętlę,** o postaci

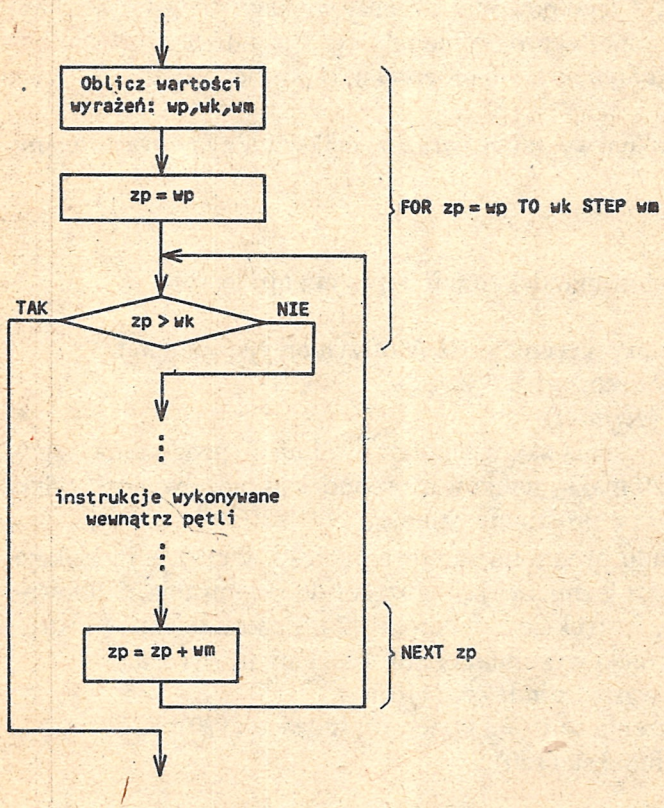
FOR zp = wp TO wk STEP wm
NEXT zp

i znaczeniu

OD zp równego *wp DO wk Z KROKIEM wm*
NASTĘPNE zp

W obydwu instrukcjach *zp* jest jednoliterową nazwą zmiennej prostej pełniącej rolę indeksu pętli, a *wp, wk, wm* są dowolnymi wyrażeniami arytmetycznymi określającymi: wartość początkową indeksu (*wp*), wartość końcową (*wk*) i przyrost wartości w kolejnych obiegach pętli (*wm*). Wykonanie instrukcji *FOR* powoduje:

— obliczenie wartości wyrażeń *wp*, *wk*, *wm* i nadanie zmiennej *zp* wartości początkowej *wp*;
— sprawdzenie, czy wartość *zp* przekracza wartość *wk*; jeżeli nie, to przejście do wykonania następnej instrukcji, jeżeli tak, to przejście do wykonania instrukcji następnej po instrukcji *NEXT* (zakończenie pętli).



Rys. 2.9. Konstrukcja pętli

Wykonanie instrukcji *NEXT* powoduje:
— zwiększenie wartości zmiennej *zp* o wartość *wm*;
— powrót na początek pętli do operacji sprawdzania, czy wartość *zp* przekracza *wk*.

Sposób użycia instrukcji *FOR* i *NEXT* ilustruje rys. 2.9.

Program sumowania elementów tablicy *X* możemy teraz zapisać

```
100 LET S = 0
110 FOR I = 1 TO N STEP 1
120 LET S = S + X(I)
130 NEXT I
```

Wartości wyrażeń *wp*, *wk* i *wm* w instrukcji *FOR* mogą być dowolne. Jeżeli wartość *wm* jest równa 1 (jak w naszym przykładzie), to człon *STEP 1* (po polsku *krok 1*) możemy opuścić.

Instrukcja *FOR* w skróconej postaci, np.

```
110 FOR I = 1 TO N
```

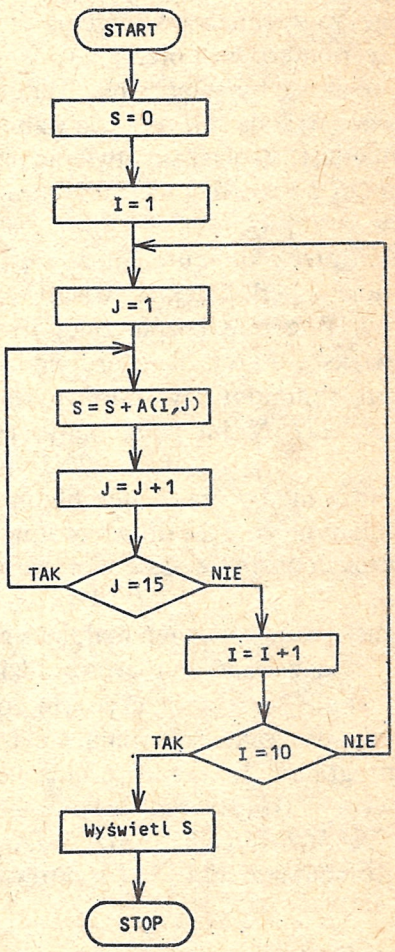
jest równoważna instrukcji

```
110 FOR I = 1 TO N STEP 1
```

Rozważmy teraz trudniejszy problem obliczenia sumy wartości wszystkich elementów dwuwymiarowej tablicy *A* (10,15). Sieć działań (rys. 2.10) pokazuje, że mamy tu dwie pętle zagłębione jedna wewnątrz drugiej. Pętla wewnętrzna dodaje wartości elementów pojedynczego wiersza tablicy, w pętli zewnętrznej są obiegane wszystkie wiersze. Odpowiedni program ma postać

```
10 DIM A (10,15)
20 S = 0
30 FOR I = 1 TO 10
40 FOR J = 1 TO 15
50 S = S + A(I,J)
60 NEXT J
70 NEXT I
80 PRINT S
```

Należy podkreślić, że pętla wewnętrzna musi być całkowicie zawarta wewnątrz pętli zewnętrznej, tzn. kolejność instrukcji *NEXT* w liniach 60 i 70 nie może zostać zmieniona. Prześledźmy działanie naszego programu. W linii 30 zmienna *I* przyjmie wartość 1. W linii 40 zmienna *J* przyjmie wartość 1 i z tak określonymi indeksami zostanie wykonane sumowanie zapisane w linii 50. W następnym obiegu pętli wewnętrznej



Rys. 2.10. Sumowanie elementów dwuwymiarowej tablicy *A(10, 15)*

zmienna *J* przyjmie wartość 2, zostanie wykonane sumowanie, itd. aż do wartości $J = 15$. Teraz indeks *I* zmieni wartość na 2 i nastąpi kolejne 15 obiegów pętli wewnętrznej ($J = 1...15$). Wykonanie obliczeń zostanie zakończone po przekroczeniu przez indeks *I* (pętla zewnętrzna) wartości 10. Obieganie pętli może zostać również przerwane przez wykonanie wewnątrz pętli instrukcji *GOTO* prowadzącej do instrukcji leżącej na zewnątrz pętli. Na przykład, jeżeli zdecydujemy się przerwać sumowanie elementów tablicy w chwili, gdy suma przekroczy wartość 50, to wystarczy uzupełnić nasz program instrukcją

55 IF S>50 THEN GOTO 80

Natomiast błędem byłoby umieszczenie w programie instrukcji

25 GOTO 50

wchodzącej do wnętrza pętli z pominięciem otwierającej instrukcji *FOR*.

Argumenty instrukcji *FOR*: *wp*, *wk* i *wm* mogą być dowolnymi stałymi, zmiennymi lub wyrażeniami arytmetycznymi, niekoniecznie o wartościach całkowitych. Używając wartości niecałkowitych należy jednak zwrócić uwagę, za warunek zakończenia pętli jest badany przez porównanie dwóch wartości obarczonych błędami (por. przykłady w p. 2.2), przez co wynik porównania może być niepewny. Dlatego w przypadku gdy zależy nam, aby pętla została wykonana *n* razy, dla wartości indeksu wp, $wp + wm$,...,$wp + (n - 1)\,wm$, nie jest bezpiecznie otwierać ją instrukcją:

FOR I = wp TO wk STEP wm

w której $wk = (n - 1)\,wm$. Zależnie od sposobu zaokrąglania wyników działań przez komputer może się okazać, że wartość indeksu po przedostatnim obiegu pętli przekroczy wartość *wk* i ostatni obieg pętli nie zostanie wykonany. Znacznie bezpieczniej jest podać jako wartość końcową liczbę nieco większą od tej, dla której pętla ma zostać wykonana po raz ostatni. Ten sam problem dotyczy porównywania wartości niecałkowitych w instrukcji *IF...THEN*. Jak pamiętamy z p. 2.2, warunek

IF(2 + 2 / 3) = (2 + 1 / 3 + 1 / 3) THEN ...

może nie być na skutek błędu obliczeń spełniony. Dlatego należy unikać w miarę możności porównywania liczb niecałkowitych, a w przypadkach koniecznych posługiwać się nierównością typu

IF ABS (X - Y) < 1E - 5 THEN ...

zachowującą pewien margines bezpieczeństwa.

Na zakończenie wprowadzimy jeszcze jedną konstrukcję programową. Jest nią **komentarz**, czyli włączony do programu tekst, objaśniający działanie tego programu, nie mający natomiast żadnego wpływu na sposób wykonania programu. Komentarze możemy umieszczać w programie po słowie kluczowym *REM* (od ang. *Remark — uwaga*), np

5 REM PROGRAM SUMUJACY ELEMENTY TABLICY

Komentarze mogą być traktowane jako element dokumentacji programu.

Tabela 2.2

Podstawowe instrukcje języka Basic

Instrukcja	Przykład	Opis działania
DATA	*DATA 4.325, 3.8E−1*	deklaracja zbioru danych
DIM	*DIM A(5),B(20,8)*	deklaracja tablic
FOR..TO..STEP	*FOR I=2 TO A+5 STEP 3*	otwarcie pętli; określenie indeksu pętli, wartości początkowej, końcowej i kroku zmiany
GOSUB	*GOSUB 1000*	skok do podprogramu
GOTO	*GOTO 45*	skok do linii programu o podanym numerze
IF..THEN	*IF A =35 THEN STOP*	instrukcja warunkowa
INPUT	*INPUT "X,Y "; X,Y*	wprowadzenie danych z klawiatury
LET	*LET A=X(I)+3*	instrukcja podstawienia
NEXT	*NEXT I*	zamknięcie pętli; modyfikacja indeksu, sprawdzenie warunków końca
PRINT	*PRINT "A+B="; A+B*	wyświetlenie wyniku na ekranie
READ	*READ A,B*	odczytanie danych ze zbioru *DATA*
REM	*REM PROGRAM*	komentarz
RESTORE	*RESTORE*	ustawienie wskaźnika odczytu ze zbioru *DATA*
RETURN	*RETURN*	powrót z podprogramu
STOP	*STOP*	przerwanie wykonania programu (z możliwością wznowienia)

Podsumowanie

— poznaliśmy podstawowe instrukcje języka *Basic* (tab.2.2):

LET zm = wa	:instrukcja podstawienia
GOTO numer	:instrukcja skoku
STOP	:zatrzymanie programu
IF warunek THEN instrukcja	:instrukcja warukowa
FOR zp = wp TO wk STEP wm	:otwarcie pętli
NEXT zp	:zamknięcie pętli
DIM lista tablic	:deklaracja tablic
REM dowolny tekst	:komentarz

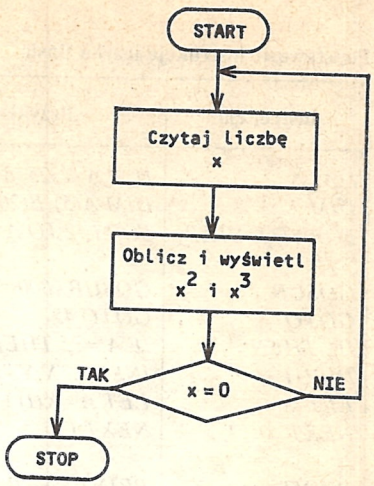
— pamiętamy, że przy porównywaniu wartości niecałkowitych mogą wystąpić błędy.

Pytania i zadania

1. Jaką postać mogą mieć elementy: *zm*, *zp*, *wa*, *wp*, *wk*, *wm*, *warunek* — wymienione w instrukcjach zamieszczonych w podsumowaniu?
2. Napisz program realizujący sieć działań z rys. 2.11.
3. Napisz programy odpowiadające sieciom działań z pytań 2, 5 i 6 w p. 2.1.
4. Napisz program obliczający wartość funkcji e^x dla $x = 0,0.1,\ldots,1$.

5. Jakie będą wartości zmiennych X i Y po wykonaniu programu

```
10 LET X = 5
20 LET Y = X + 2
30 LET Y = Y * 2 - 2
40 REM X MA WARTOSC 4, Y = 5
```



Rys. 2.11. Rysunek do zadania 2. Sieć działań programu obliczającego kwadraty i sześciany liczb

2.4
Wprowadzanie danych do programu

W rozdziale 1 poznaliśmy możliwość odczytania danych z klawiatury w czasie wykonania programu. Służy do tego **instrukcja wprowadzania danych *INPUT*** o postaci

INPUT "tekst"; lista argumentów

w której *"tekst"* oznacza dowolną stałą tekstową, a *lista argumentów* zawiera nazwy dowolnych zmiennych, których wartości mają być wprowadzone. Stała *"tekst"* może być opuszczona, jeżeli jednak wystąpi, to jest wyświetlana na ekranie przed rozpoczęciem wprowadzania danych.

W języku *Basic* istnieje również druga często dogodniejsza możliwość określania danych przez utworzenie zbioru danych, dołączonego do programu. **Instrukcja utworzenia zbioru danych *DATA*** o postaci

DATA w1, w2,...,wn

definiuje zbiór wartości *w1*, *w2*,...,*wn*, które mogą być następnie czytane w tej samej kolejności, w jakiej są wymienione w instrukcji *DATA*. Jeżeli w dalszej części programu występują inne instrukcje *DATA*, to wymienione w nich dane zostają dopisane do już istniejącego w programie zbioru danych. Tak więc fragment programu

```
10 DATA 1,2
20 DATA 3
80 DATA 4,5,6
```

definiuje zbiór danych zawierający liczby: 1,2,3,4,5,6.

Instrukcja pobrania danych *READ* o postaci

READ z1, z2,...,zm

powoduje pobranie *m* kolejnych wartości ze zbioru danych i podstawienie ich jako wartości wymienionych w instrukcji zmiennych: *z1*, *z2*,...,*zm*. Jeżeli w programie oprócz trzech wymienionych instrukcji *DATA* wystąpią dwie instrukcje *READ*

30 READ A, B, C
40 READ X(1), X(2), X(3)

to w rezultacie poszczególnym zmiennym zostaną przypisane wartości

$A = 1$, $B = 2$, $C = 3$, $X(1) = 4$, $X(2) = 5$, $X(3) = 6$

Zauważmy, że instrukcje *DATA* definiujące zbiór danych nie muszą wystąpić w programie przed instrukcjami *READ* czytającymi dane ze zbioru. Ważne jest tylko to, aby wystąpiły one w programie i aby liczba wymienionych w nich danych nie była mniejsza od liczby zmiennych występujących w instrukcji *READ*.

Kolejność czytania danych ze zbioru może być zmieniona przez użycie **instrukcji inicjalizacji zbioru *RESTORE*** o postaci

RESTORE numer

w której *numer* wskazuje jedną z instrukcji *DATA* umieszczonych w programie. Kolejne instrukcje *READ* wykonane po instrukcji *RESTORE* spowodują odczytywanie danych od pierwszej danej wymienionej we wskazanej instrukcji *DATA*. Pominięcie *numeru* w instrukcji *RESTORE* spowoduje rozpoczęcie czytania od początku zbioru danych zdefiniowanego w programie.

Tak więc wykonanie programu

10 DATA 1,2
20 READ A,B
30 RESTORE
40 READ C

spowoduje nadanie zmiennym wartości

$A = 1$, $B = 2$, $C = 1$

Próba wykonania programu

10 DATA 1,2
20 READ A,E
40 READ C

spowoduje zasygnalizowanie błędu, gdyż dla operacji *READ C* zabraknie wartości w dwuelementowym zbiorze danych.

Natomiast wykonanie programu

10 READ A,B,C
20 RESTORE 50
30 READ X,Y
40 DATA 1,2
50 DATA 3,4

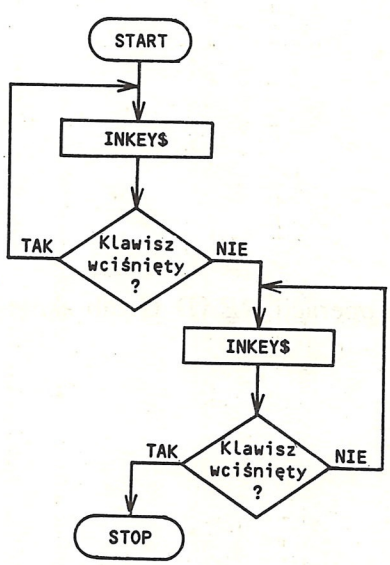
spowoduje nadanie zmiennym wartości

$A = 1$, $B = 2$, $C = 3$, $X = 3$, $Y = 4$

Powróćmy do wprowadzania danych z klawiatury. Działanie instrukcji *INPUT* jest bardzo złożone. Nie tylko czyta ona dane, ale również powoduje wyświetlenie wszystkich pisanych znaków na ekranie. Co więcej, działanie instrukcji trwa aż do chwili naciśnięcia klawisza *ENTER*. W większości typowych przypadków ten sposób wprowadzania danych jest bardzo dogodny, niekiedy jednak stwarza zbyt duże ograniczenia. Na przykład instrukcji *INPUT* nie możemy stosować, jeżeli chcemy na bieżąco analizować pisane znaki, bez czekania na naciśnięcie klawisza *ENTER*, jak również w przypadku gdy nie chcemy, aby pisane znaki były widoczne na ekranie. W obydwu wymienionych sytuacjach możemy natomiast stosować **funkcję *INKEY$*** natychmiast odczytującą pojedynczy znak z klawiatury (bez wyświetlania tego znaku na ekranie).

Użycie funkcji *INKEY$* powoduje sprawdzenie stanu klawiatury i podstawienie jako wartości funkcji jednoznakowego tekstu zawierającego napisany na klawiaturze znak. Jeżeli żaden klawisz nie został naciśnięty, to wartością funkcji staje się tekst pusty. Korzystając z funkcji *INKEY$* musimy pamiętać o konieczności odczekania (w programie) na pojawienie się znaku.

Szczegóły działania funkcji *INKEY$* zależą od sposobu dołączenia klawiatury i są różne w różnych komputerach. W *Spectrum* każde użycie funkcji *INKEY$* powoduje odczytanie aktualnie naciśniętego znaku. Związane jest z tym niebezpieczeństwo wielokrotnego odczytania tego samego znaku, występujące w przypadkach, gdy nasz program odczytuje kilka kolejnych znaków. Może się bowiem zdarzyć, że próba odczytu następnego znaku nastąpi zanim zwolnimy klawisz poprzedni. Spowoduje to powtórne odczytanie już raz odczytanego znaku. Aby odczytać każdy napisany znak dokładnie jeden raz musimy napisać program realizujący następujący algorytm: najpierw zaczekać na zwolnienie poprzednio naciśniętego klawisza, następnie zaczekać na naciśnięcie nowego klawisza i dopiero teraz odczytać napisany znak. Sieć działań tego algorytmu przedstawiono na rys. 2.12, a odpowiadający jej program ma postać



Rys. 2.12. Odczytywanie pojedynczego znaku

```
100 IF INKEY$ < >"" THEN GOTO 100
110 IF INKEY$ = "" THEN GOTO 110
120 LET Z$ = INKEY$
```

Symbol "" oznacza tekst pusty. Po wykonaniu programu wartością zmiennej *Z$* jest tekst złożony z napisanego znaku.

W mikrokomputerze *Meritum* funkcja *INKEY$* działa nieco inaczej. Klawiatura tego mikrokomputera jest konstrukcyjnie zabezpieczona przed wielokrotnym odczytaniem znaku raz naciśniętego klawisza. Naty-

chmiast po odczytaniu znaku jest on (wewnątrz klawiatury) kasowany. Ponowne użycie funkcji *INKEY$* da w odpowiedzi łańcuch pusty, pomimo przytrzymywania naciśniętego klawisza. Program odczytania znaku zmienia się do postaci

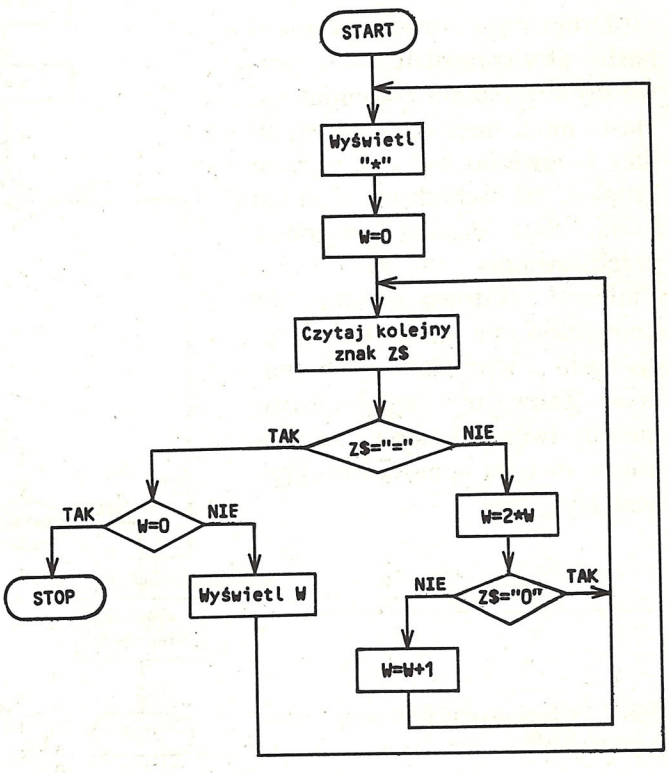
```
100 Z$ = INKEY$
110 IF Z$ = "" THEN GOTO 100
```

Na zakończenie napiszemy dwa programy ilustrujące wykorzystanie funkcji *INKEY$*. Pierwszy z nich oblicza wartość liczby dwójkowej $b_n...b_1b_0$ i podaje ją w zapisie dziesiętnym (por. pytanie 4 , p. 2.1), natomiast drugi zabezpieczy nasze programy przed uruchomieniem ich przez niepowołanego użytkownika.

Program pierwszy

Przyjmiemy, że program powoduje wyświetlenie znaku gwiazdki, po którym możemy wprowadzić dowolny ciąg bitów zakończony znakiem = (równa się). Bezpośrednio za tym znakiem powinna zostać wyświetlona liczba w zapisie dziesiętnym, po czym w nowej linii mikrokomputer wyświetla następną gwiazdkę i oczekuje na podanie kolejnej liczby. Działanie programu kończy wprowadzenie liczby zero. Przy opracowaniu algorytmu posłużymy się wzorem sprowadzającym liczbę daną w zapisie dwójkowym do postaci dziesiętnej

$$w = b_0 + 2b_1 + 2^2b_2 + \ldots + 2^nb_n =$$
$$= b_0 + 2\,(b_1 + 2\,(b_2 + 2\,(\ldots + 2\,(b_{n-1} + 2b_n)\ldots)))$$



Rys. 2.13. Algorytm wprowadzania liczby dwójkowej

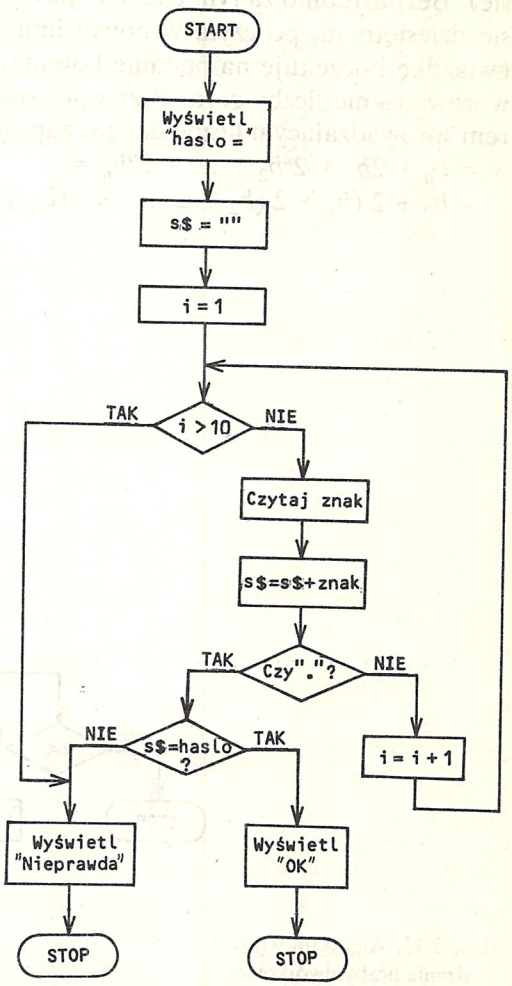
Korzystając z tego wzoru możemy narysować sieć działań (rys. 2.13) i napisać program (w wersji dla mikrokomputera *Meritum*)

```
 10 REM OBLICZANIE WARTOSCI LICZBY DWOJKOWEJ
 20 PRINT "*";
 30 LET W = 0
 40 LET Z$ = INKEY$
 50 IF Z$ = "" THEN GOTO 40
 60 PRINT Z$;
 70 IF Z$ = "=" THEN GOTO 200
 80 LET W = 2 * W
 90 IF Z$ = "0" THEN GOTO 40
100 LET W = W + 1
110 GOTO 40
200 PRINT W
210 IF W = 0 THEN STOP
220 GOTO 20
```

Instrukcja *PRINT Z$*; w linii 60 powoduje wyświetlanie na ekranie wprowadzanych bitów liczby dwójkowej.

Program drugi

Drugi program zapyta nas o hasło, aby sprawdzić nasze prawo do korzystania z komputera. Hasło może mieć co najwyżej 10 liter i powinno być zakończone kropką, też wchodzącą w skład hasła. Sieć działań programu przedstawiono na rys. 2.14. Zmienna tekstowa *s$* służy do kompletowania słowa tworzącego hasło z wprowadzanych znaków. Zauważmy podobieństwo metody tworzenia słowa ze znaków z metodą sumowania ciągu liczb (rys. 2.3).



Rys. 2.14. Sieć działań programu kontrolującego hasło

Program w wersji dla mikrokomputera *Spectrum*

```
10 REM Sprawdzenie hasla
20 PRINT "Haslo=";
30 LET s$ = ""
40 FOR i = 1 TO 10
50 IF INKEY$ <>"" THEN GOTO 50
60 IF INKEY$ = "" THEN GOTO 60
70 LET s$ = s$ + INKEY$
80 IF INKEY$ = "." THEN GOTO 200
90 NEXT i
100 PRINT " *** Nieprawda *** "
110 STOP
200 READ h$
210 IF s$ <> h$ THEN GOTO 100
220 PRINT " * * * OK * * *
230 GOTO 250
240 DATA "KOMPUTER."
```

Oczywiście, hasło pisane na klawiaturze nie pojawia się na ekranie.

Podsumowanie

— poznaliśmy składnię instrukcji wprowadzania danych:

DATA w1,w2,...,wn	: deklaracja zbioru danych
READ z1,z2,...,zm	: pobranie danych
RESTORE numer	: ustawienie wskaźnika zbioru

— poznaliśmy funkcję *INKEY$* odczytującą znak z klawiatury.

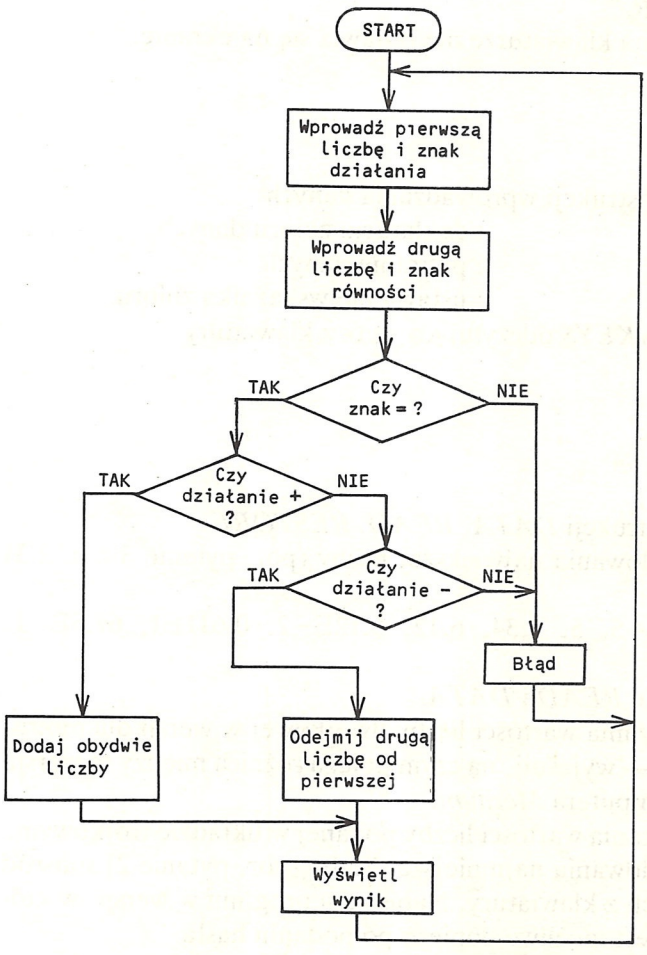
Pytania i zadania

1. Wyjaśnij działanie instrukcji *DATA*, *READ*, *RESTORE*.
2. Napisz program znajdowania największej liczby (por. pytanie 3 z p. 2.3) spośród:
   1.2, 0.125E1, 1234E−5, 5, 5.34, 6.12, 6.12E−2, 0.611E1, 61.5E−1, 0.613E1
   Wykorzystaj instrukcje *READ* i *DATA*
3. Napisz program obliczania wartości liczby dwójkowej w wersji dla mikrokomputera *Spectrum* — wyjaśnij, na czym polega różnica między tą wersją a wersją dla mikrokomputera *Meritum*.
4. Napisz program obliczania wartości liczby podanej w układzie trójkowym.
5. Napisz program znajdowania najmniejszej liczby (por. pytanie 2) spośród *n* liczb wprowadzonych z klawiatury. Napisz ten program w wersji, w której użycie programu jest możliwe dopiero po podaniu hasła.

2.5
Podprogramy. Przykład programu

Napiszemy program realizujący obliczenia na liczbach wprowadzanych i wyświetlanych w zapisie rzymskim. Ograniczymy nasz program do liczb naturalnych (Rzymianie nie znali ułamków, liczb ujemnych ani zera), nie większych niż 100. Działanie programu powinno wyglądać następująco. Po uruchomieniu programu piszemy działanie, np.
XV + VII =
a komputer dopisuje wynik
XV + VII = XXII
po czym wraca do stanu oczekiwania na wprowadzenie następnego działania. Dopuszczalne są operacje dodawania (+) i odejmowania (−).

Ogólny schemat programu jest następujący (rys. 2.15): wprowadzamy ko-



Rys. 2.15. Ogólna sieć działań „rzymskiego kalkulatora"

lejno obydwa argumenty i znak działania, wykonujemy działanie i wyświetlamy wynik. Znak operacji różny od *plus* lub *minus* powoduje zasygnalizowanie błędu. Teraz zaczynają się trudności. Rzymski zapis nie jest dobrze przystosowany do wykonywania działań arytmetycznych. Wprowadzając liczbę musimy więc obliczyć jej wartość w zapisie dziesiętnym, wykonać działanie, a następnie znów przekodować wynik do zapisu rzymskiego. Główna trudność programu leży w konwersji liczb, tzn. w zmianie formy zapisu: z rzymskiego na dziesiętny i odwrotnie.

Reguły obliczania wartości liczb rzymskich są ogólnie znane. W interesującym nas zakresie liczb podstawowymi znakami (cyframi rzymskimi) są:

I = 1
V = 5
X = 10
L = 50
C = 100

Liczba jest ciągiem cyfr (rzymskich) zapisanych od strony lewej do prawej w zasadzie w kolejności malejących wartości cyfr. Wartość tak zapisanej liczby obliczamy jako sumę wartości poszczególnych cyfr, np.

LXXVII = 50 + 10 + 10 + 5 + 1 + 1 = 77

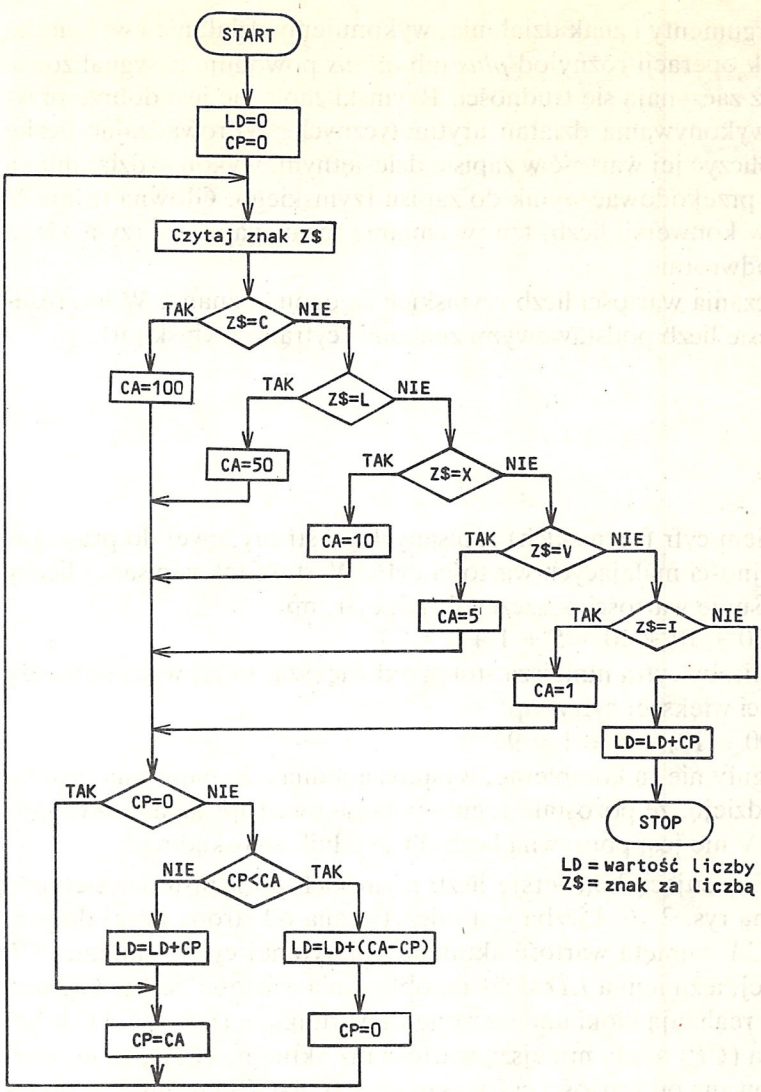
Jeżeli w zapisie liczby cyfra mniejsza stoi przed większą, to jej wartość należy odjąć od wartości większej cyfry, np.

XCVI = (100 − 10) + 5 + 1 = 96

Wymienione reguły nie są kompletne, wystarczą jednak do napisania programu, a mamy nadzieję, że pozostałe reguły porządkowe (np. taką, że III równa się 3, ale VVV nie jest poprawną liczbą) Czytelnik zna skądinąd.

Sieć działań opisującą konwersję liczb rzymskich do zapisu dziesiętnego przedstawiono na rys. 2.16. Liczba jest odczytywana od strony lewej do prawej. Zmienna *CA* pamięta wartość aktualnie odczytanej cyfry, zmienna *CP* cyfry poprzedniej, a zmienna *LD* służy do obliczania wartości liczby. Główną część algorytmu realizują bloki umieszczone za warunkiem ($CP < CA$). Jeżeli cyfra poprzednia (*CP*) miała mniejszą wartość niż aktualna (*CA*), to jej wartość jest odejmowana od wartości cyfry aktualnej *(CA-CP)* i wynik jest dodawany do wartości liczby ($LD = LD + (CA - CP)$). W przeciwnym przypadku wartość cyfry poprzedniej (*CP*) jest dodawana do wartości liczby ($LD = LD + CP$), a wartość cyfry aktualnej staje się wartością poprzednią (dla następnego obiegu pętli). Sytuacja: $CP = 0$ oznacza brak cyfry poprzedniej na początku programu oraz po obliczeniu wartości przestawionej pary cyfr. Zakończenie programu następuje po odczytaniu znaku nie będącego cyfrą rzymską. Po zakończeniu, wartość zmiennej *LD* określa wartość wprowadzonej liczby, a zmienna tekstowa *Z$* pamięta znak wprowadzony bezpośrednio za odczytaną liczbą.

Zmiana postaci liczby z zapisu dziesiętnego na zapis rzymski jest znacznie prostsza. Wykonamy ją według prymitywnego algorytmu przedstawionego na rys. 2.17. Algorytm ten działa prawidłowo dla liczb z zakresu 1...299.



Rys. 2.16. Obliczanie wartości liczby rzymskiej

Ostatnim problemem naszego programu jest problem odczytania znaku wprowadzanego z klawiatury w czasie wczytywania liczby rzymskiej. Ponieważ chcemy przetwarzać kolejne znaki na bieżąco, bez użycia klawisza *ENTER* nie możemy użyć instrukcji *INPUT*. Musimy wykorzystać funkcję *INKEY$* w sposób opisany siecią działań z rys. 2.12.

Metoda zastosowana przez nas przy analizie problemu realizacji „rzymskiego kalkulatora" polegająca na stopniowym rozbijaniu zadania na coraz prostsze fragmenty nazywana jest metodą **zstępującą** (ang. *top-down*), od pojęć ogólniejszych do coraz bardziej szczegółowych. Przy pisaniu programu zastosujemy podejście odwrotne, nazywane **wstępującym** (ang. *bottom-up*), pole-

gające na budowaniu coraz większych jednostek programu, poczynając od jednostek najprostszych.

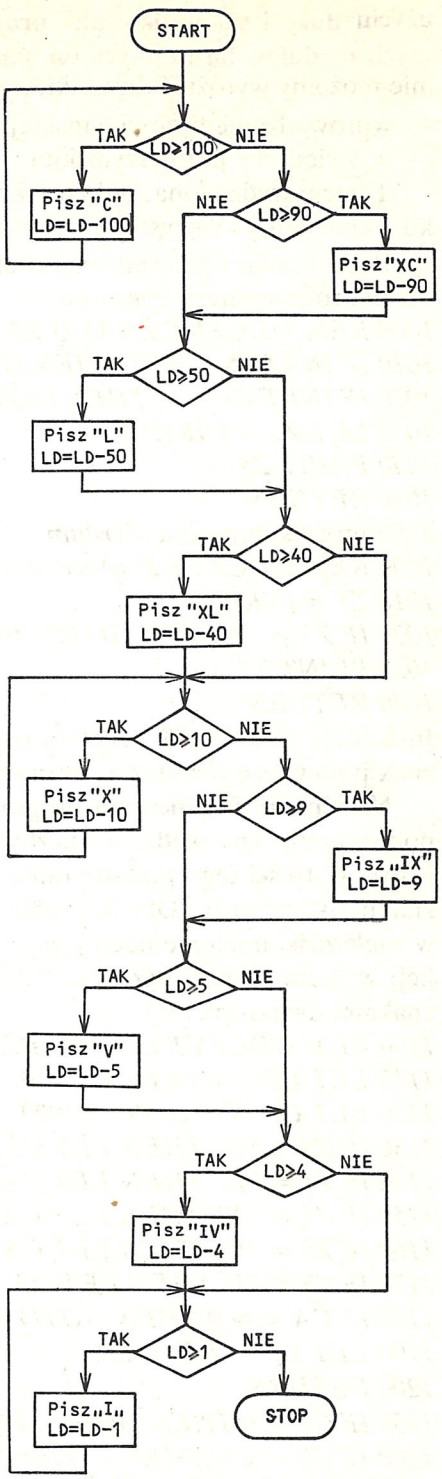
Program odczytywania znaku przy użyciu funkcji *INKEY$* napisaliśmy już poprzednio. Zajmiemy się więc od razu następnym większym fragmentem wprowadzania liczby rzymskiej. Spójrzmy najpierw na ogólną sieć działań programu (rys. 2.15). Czynność wprowadzania liczby rzymskiej występuje tu w dwóch blokach, co oznacza, że identyczny program wprowadzania liczby będziemy musieli napisać dwukrotnie, co znacznie wydłuży zapis całego programu. Znacznie dogodniej byłoby napisać program wprowadzania tylko jeden raz, a potem powiedzieć *zrób jeszcze raz to samo*. Możliwość taką dają w językach programowania podprogramy.

Podprogram jest wyodrębnionym fragmentem programu, umieszczonym w programie głównym w jednym miejscu, lecz możliwym do wywołania (wykonania) w wielu innych miejscach programu. **Wywołanie podprogramu** jest dokonywane instrukcją ***GOSUB*** (co stanowi skrót od *Go to subroutine — idź do podprogramu)* o postaci

GOSUB numer

Wykonanie instrukcji *GOSUB* powoduje skok do linii o podanym *numerze*, z jednoczesnym zapamiętaniem numeru linii następnej po instrukcji *GOSUB*. W ten sposób rozpoczyna się wykonanie podprogramu, który musi kończyć się **instrukcją powrotu *RETURN*** (co znaczy *wróć*) powodującą powrót do linii o zapamiętanym w chwili wywołania numerze

Stosowanie podprogramów daje również inne korzyści poza skróceniem długości programu. Dzięki ich



Rys. 2.17. Konwersja liczby dziesiętnej do zapisu rzymskiego

użyciu **duży i skomplikowany program zostaje podzielony na szereg mniejszych modułów łatwiejszych do napisania i uruchomienia.** W naszym programie możemy wyróżnić dwa takie wydzielone moduły:

— wprowadzenie liczby rzymskiej,
— wyświetlenie liczby rzymskiej.

Trzecią wydzieloną, dobrze określoną czynnością jest wprowadzanie znaku z klawiatury. Wszystkim trzem modułom nadamy postać podprogramów.

Podprogram wprowadzenia znaku może mieć postać:
dla mikrokomputera *Spectrum*

```
1000 REM ODCZYT ZNAKU Z$ Z KLAWIATURY
1010 IF INKEY$ < > "" THEN GOTO 1010
1020 IF INKEY$ = "" THEN GOTO 1020
1030 LET Z$ = INKEY$
1040 PRINT Z$ ;
1050 RETURN
```

a dla mikrokomputera *Meritum*

```
1000 REM ODCZYT ZNAKU Z$ Z KLAWIATURY
1010 Z$ = INKEY$
1020 IF Z$ = "" THEN GOTO 1010
1030 PRINT Z$;
1040 RETURN
```

Instrukcja *PRINT Z$* ; została dodana po to, aby napisany na klawiaturze znak pojawił się również na ekranie.

Następnym fragmentem programu kalkulatora, który napiszemy, będzie podprogram wprowadzania liczby rzymskiej z klawiatury (algorytm z rys. 2.16). W treści tego podprogramu wykorzystamy podprogram odczytywania znaku z klawiatury. Dla skrócenia zapisu programu wykorzystamy dostępną w większości implementacji języka *Basic* możliwość zapisywania kilku instrukcji w jednej linii programu. Tak zapisane instrukcje oddzielamy od siebie znakami dwukropka (:).

```
1100 REM ODCZYT LICZBY RZYMSKIEJ LD
1110 LET LD = 0 : LET CP = 0
1120 LET CA = 0: GOSUB 1000
1130 IF Z$ = "C" THEN LET CA = 100
1140 IF Z$ = "L" THEN LET CA = 50
1150 IF Z$ = "X" THEN LET CA = 10
1160 IF Z$ = "V" THEN LET CA = 5
1170 IF Z$ = "I" THEN LET CA = 1
1180 IF CA < > 0 THEN GOTO 1250
1190 LET LD = LD + CP
1200 RETURN
1250 IF CP = 0 THEN GOTO 1280
1260 IF CP < CA THEN GOTO 1300
1270 LET LD = LD + CP
```

```
1280 LET CP = CA : GOTO 1120
1300 LET LD = LD + CA - CP
1310 LET CP = 0 : GOTO 1120
```

Zauważmy przy okazji, że instrukcja *RETURN* nie musi być ostatnią napisaną instrukcją w tekście podprogramu. Musi być natomiast ostatnią wykonaną instrukcją podprogramu.

Kolejny podprogram realizuje algorytm wyświetlania liczby w zapisie rzymskim (rys. 2.17). Mimo swej długości podprogram jest tak prosty, że nie wymaga komentarza.

```
1400 REM WYSWIETLANIE LICZBY RZYMSKIEJ LD
1410 IF LD < 100 THEN GOTO 1450
1420 PRINT "C"; : LET LD = LD - 100
1430 GOTO 1410
1450 IF LD < 90 THEN GOTO 1470
1460 PRINT "XC"; : LET LD = LD - 90
1470 IF LD < 50 THEN GOTO 1490
1480 PRINT "L"; : LET LD = LD - 50
1490 IF LD < 40 THEN GOTO 1510
1500 PRINT "XL"; : LET LD = LD - 40
1510 IF LD < 10 THEN GOTO 1550
1520 PRINT "X"; : LET LD = LD - 10
1530 GOTO 1510
1550 IF LD < 9 THEN GOTO 1570
1560 PRINT "IX"; : LET LD = LD - 9
1570 IF LD < 5 THEN GOTO 1590
1580 PRINT "V"; : LET LD = LD - 5
1590 IF LD < 4 THEN GOTO 1610
1600 PRINT "IV"; : LET LD = LD - 4
1610 IF LD < 1 THEN RETURN
1620 PRINT "I"; : LET LD = LD - 1
1630 GOTO 1610
```

Należy zwrócić uwagę na konieczność użycia średnika w instrukcjach *PRINT*, dla zapewnienia wydruku kolejnych znaków bezpośrednio po sobie.

I wreszcie program główny. Dzięki wykorzystaniu przygotowanych zawczasu podprogramów będzie on krótki i zrozumiały.

```
10 REM RZYMSKI KALKULATOR
20 GOSUB 1100 : REM PIERWSZY ARGUMENT
30 IF Z$ = "." THEN STOP
40 LET X = LD : LET A$ = Z$
50 GOSUB 1100 : REM DRUGI ARGUMENT
60 IF Z$ < > "=" THEN GOTO 200
70 IF A$ < > "+" THEN GOTO 90
80 LET LD = X + LD : GOTO 150
90 IF A$ < > "-" THEN GOTO 200
```

100 LET LD = X − LD
110 IF LD < = 0 THEN GOTO 200
150 GOSUB 1400 : REM WYSWIETLENIE WYNIKU
160 PRINT : GOTO 20
200 PRINT " ?" : GOTO 20

```
10 REM RZYMSKI KALKULATOR
20 GOSUB 1100
30 IF Z$="." THEN STOP
40 LET X=LD : LET A$=Z$
50 GOSUB 1100
60 IF Z$<>"=" THEN GOTO 200
70 IF A$<>"+" THEN GOTO 90
80 LET LD=X+LD : GOTO 150
90 IF A$<>"-" THEN GOTO 200
100 LET LD=X-LD
110 IF LD<=0 THEN GOTO 200
150 GOSUB 1400
160 PRINT : GOTO 20
200 PRINT "   ?" : GOTO 20
1000 REM ODCZYT ZNAKU Z$ Z KLAWIATURY
1001 Z$=INPUT$(1):PRINT Z$;:RETURN
1010 IF INKEY$<>"" THEN GOTO 1010
1020 IF INKEY$="" THEN GOTO 1020
1030 LET Z$=INKEY$
1040 PRINT Z$;
1050 RETURN
1100 REM ODCZYT LICZBY RZYMSKIEJ LD
1110 LET LD=0 : LET CP=0
1120 LET CA=0 : GOSUB 1000
1130 IF Z$="C" THEN LET CA=100
1140 IF Z$="L" THEN LET CA=50
1150 IF Z$="X" THEN LET CA=10
1160 IF Z$="V" THEN LET CA=5
1170 IF Z$="I" THEN LET CA=1
1180 IF CA<>0 THEN GOTO 1250
1190 LET LD=LD+CP
1200 RETURN
1250 IF CP=0 THEN GOTO 1280
1260 IF CP<CA THEN GOTO 1300
1270 LET LD=LD+CP
1280 LET CP=CA : GOTO 1120
1300 LET LD=LD+CA-CP
1310 LET CP=0 : GOTO 1120
1400 REM WYSWIETLENIE LICZBY RZYMSKIEJ LD
1410 IF LD<100 THEN GOTO 1450
1420 PRINT "C"; : LET LD=LD-100
1430 GOTO 1410
1450 IF LD<90 THEN GOTO 1470
1460 PRINT "XC"; : LET LD=LD-90
1470 IF LD<50 THEN GOTO 1490
1480 PRINT "L"; : LET LD=LD-50
1490 IF LD<40 THEN GOTO 1510
1500 PRINT "XL"; : LET LD=LD-40
1510 IF LD<10 THEN GOTO 1550
1520 PRINT "X"; : LET LD=LD-10
1530 GOTO 1510
1550 IF LD<9 THEN GOTO 1570
1560 PRINT "IX"; : LET LD=LD-9
1570 IF LD<5 THEN GOTO 1590
1580 PRINT "V"; : LET LD=LD-5
1590 IF LD<4 THEN GOTO 1610
1600 PRINT "IV"; : LET LD=LD-4
1610 IF LD<1 THEN RETURN
1620 PRINT "I"; : LET LD=LD-1
1630 GOTO 1610
```

Rys. 2.18. Tekst programu „rzymskiego kalkulatora"

Dodatkowa instrukcja w linii 30 daje możliwość zakończenia programu przez napisanie kropki (.) na początku linii. Pełny tekst programu wydrukowany przez drukarkę pokazano na rys. 2.18.

Podsumowanie

— poznaliśmy zastosowanie podprogramów: wydzielonach modułów programu możliwych do wielokrotnego wywołania; instrukcje:
GOSUB numer,
RETURN;
— poznaliśmy dwa sposoby analizy problemu i syntezy programu: zstępujący (*top-down*) i wstępujący (*bottom-up*);
— wiemy, że w jednej linii programu możemy zapisać kilka instrukcji oddzielonych znakami dwukropka (:).

Pytania i zadania

1. Wyjaśnij pojęcie podprogramu.
2. Omów instrukcje
GOSUB numer
RETURN.
3. Najprostszy algorytm zamiany liczby dziesiętnej D (naturalnej) na dwójkową jest następujący:
   a) podziel liczbę D przez dwa, zapamiętaj resztę (**0** lub **1**) jako najmniej znaczący (kolejny) bit liczby, a iloraz podstaw jako nową wartość D;
   b) jeżeli $D = 0$, to koniec, jeżeli nie, to przejdź do punktu a).
   Narysuj sieć działań programu konwersji (zmiany) dziesiętno-dwójkowej.
4. Napisz podprogramy:
   — wprowadzania i obliczania wartości liczby dwójkowej (por. pyt. 4 w p. 2.1);
   — wyświetlanie liczby w zapisie dwójkowym (por. pyt. 3).
   Napisz program kalkulatora działającego w zapisie dwójkowym.

2.6
Niskorozdzielcza grafika mikrokomputera Meritum

Podstawowe instrukcje służące do wprowadzania i wyprowadzania liczb i tekstów, tzn.: *PRINT*, *INPUT* i *READ* są zdefiniowane w standardzie języka *Basic* i działają identycznie we wszystkich komputerach. Oprócz tych instrukcji każdy komputer wykonuje zestaw dodatkowych, niestandardowych instrukcji wejścia-wyjścia związanych ze specyficznymi właściwościami wykorzystywanych urządzeń. Przykładem takich instrukcji mogą być instrukcje graficzne pozwalające na wyświetlanie rysunków na ekranie telewizora. Różnice między możliwościami różnych mikrokomputerów są w tym zakresie duże, ze względu na różną organizację wyświetlania obrazu.

Przy wyświetlaniu liczb i tekstów ekran monitora mikrokomputera *Meritum* jest podzielony na 16 linii po 64 znaków każda. Łączna liczba pozycji znakowych na ekranie wynosi więc 1024. Każda pozycja znakowa ma swój numer, przy czym pozycje w pierwszej linii są numerowane z lewa na prawo od 0 do 63, w drugiej linii od 64 do 127, itd., aż do ostatniej linii, w której pozycje są numerowane od 960 do 1023. Jeżeli umówimy się numerować linie od 0 do 15 i kolumny od 0 do 63, to numer pozycji znakowej w *I*-tej linii i *J*-tej kolumnie jest dany wzorem

$P = 64 * I + J$

Numery pozycji mogą być wykorzystane do wskazania miejsca wyświetlania danego znaku na ekranie monitora. Służy do tego instrukcja *PRINT* z **funkcją** @, o postaci

PRINT @ pozycja, lista elementów

w której *lista elementów* może być zapisana w taki sam sposób jak w zwykłej instrukcji *PRINT*, a wartość wyrażenia *pozycja* określa miejsce na ekranie, w którym zostanie wyświetlony pierwszy znak, czyli od której rozpocznie się wyświetlanie.

Na przykład program

```
10 CLS
20 PRINT @ 350, "A"
30 PRINT @ 413, "A   A"
40 PRINT @ 476, "A      A"
50 PRINT @ 540, "AAAAA"
60 PRINT @ 604, "A      A"
70 PRINT @ 668, "A      A"
80 GOTO 80
```

spowoduje wyświetlenie pośrodku pustego ekranu wielkiej litery *A* zbudowanej z liter *A* normalnego rozmiaru.

Przy redagowaniu informacji wyświetlanej na ekranie użyteczna jest też

funkcja *TAB* przesuwająca w obrębie jednej linii miejsce wyświetlania znaku. Argumentem funkcji *TAB* jest numer kolumny wskazujący miejsce następnego wyświetlanego znaku. W poprzednim programie linia 70 powodowała wyświetlenie pierwszej litery *A* w kolumnie 28, a drugiej w kolumnie 32. Działanie programu nie zmieni się, jeżeli zastąpimy poprzednią linię 70 nową

70 PRINT @ 668, "A"; TAB (32) "A"

Rys. 2.19. Znak graficzny mikrokomputera *Meritum*

W instrukcji *PRINT*, oprócz typowych znaków pisarskich, mogą być też używane specjalne znaki graficzne (które mogą również być włączane do zmiennych tekstowych), pozwalające na sporządzanie prostych rysunków. Każdy **znak graficzny** zajmuje na ekranie jedną pozycję znakową. Jest on zbudowany z sześciu punktów (rys. 2.19), z których każdy może być jasny (niewidoczny na tle ekranu) lub ciemny. 64 znaki graficzne o kodach z zakresu 128...191 odpowiadają wszystkim możliwym kombinacjom punktów jasnych i ciemnych. Kod znaku może być obliczony na podstawie wzorca kodowania pokazanego na rys. 2.19. Również typowe znaki pisarskie mają swoje kody, przy czym mikrokomputer *Meritum* wykorzystuje znormalizowany system kodowania określony przez międzynarodowy standard *ASCII*. Zgodnie z tym standardem kody 0...31 odpowiadają rozmaitym znakom sterującym sposobem wyświetlania, kody 32...127 odpowiadają znakom widocznym na ekranie (*Meritum* nie wykorzystuje wszystkich kodów), a kody powyżej 127 mogą być wykorzystane dowolnie. Aby wykorzystać kod znaku, np. w instrukcji *PRINT*, należy najpierw zamienić go na jednoznakowy tekst, zawierający znak odpowiadający temu kodowi. Zadanie to wykonuje **funkcja *CHR$***. Na przykład program

```
10 CLS: PRINT : PRINT : PRINT
20 FOR I = 0 TO 9
30 FOR J = 0 TO 15
40 PRINT CHR$ (16 * I + J + 32);
50 NEXT J
60 PRINT
70 NEXT I
```

spowoduje wyświetlenie w 10 liniach i 16 kolumnach wszystkich znaków odpowiadających kodom 32...191.

Traktując punkt (znaku graficznego) jako najmniejszy element ekranu możemy uważać, że cała powierzchnia ekranu jest podzielona na 6144 punkty ułożone w 48 wierszach (numerowanych z góry na dół od 0 do 47) i 128 kolu-

mnach (numerowanych od 0 do 127). Stan każdego punktu może być ustalony albo przez wykonanie instrukcji *PRINT* wyświetlającej znaki graficzne, albo indywidualnie przy użyciu jednej z dwóch **instrukcji zmiany stanu punktu**

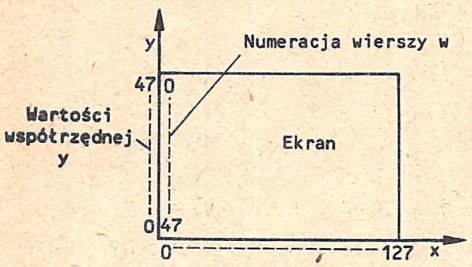
SET (w,x) — zapal

RESET (w,x) — zgaś

Parametrami obydwu instrukcji są współrzędne (*w* — numer wiersza, *x* — numer kolumny) zapalanego lub gaszonego punktu ekranu. Stan każdego punktu możemy też sprawdzić przy użyciu **funkcji badania stanu**

POINT (w, x)

której wartością jest 0, jeżeli punkt jest „zgaszony" (jasny), lub 1, jeżeli punkt jest „zapalony" (ciemny).



Rys. 2.20. Ekran mikrokomputera *Meritum* jako obraz pierwszej ćwiartki układu współrzędnych *x*0*y*

Zaciemniając odpowiednie punkty możemy rysować na ekranie nieskomplikowane obrazki. W większości przypadków będziemy traktować ekran jako obraz pierwszej ćwiartki kartezjańskiego układu współrzędnych *x*0*y*. Pewną niewygodę stanowi sposób adresacji punktów ekranu: w pionie numeracja punktów wzrasta z góry na dół, przeciwnie niż wartość współrzędnej *y* układu (rys. 2.20). Przy kreśleniu rysunków musimy pamiętać o konieczności przeliczania współrzędnej *y* rysowanych punktów na numer wiersza *w*

$$w = 47 - y$$

stanowiący argument instrukcji *SET* i *RESET*. Na przykład, korzystając z równania prostej

$$y = kx$$

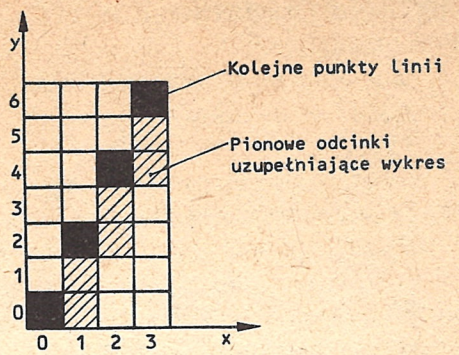
możemy napisać program rysujący odcinek o zadanym nachyleniu *k*.

```
10 INPUT "NACHYLENIE =" ; K
20 CLS
30 FOR X = 0 TO 127
40 Y = INT (K * X)
50 IF Y>47 THEN GOTO 10
60 SET (X, 47 - Y)
70 NEXT X
80 GOTO 10
```

Program działa w miarę dobrze tylko dla wartości współczynnika *k* nie większych niż 1. Dla wartości większych z odcinka zostaje zaledwie kilka punktów. Przyczyną jest duży skok zmiennej *y*, który dla uzyskania ciągłości linii należy wypełnić punktami (rys. 2.21), zmieniając nasz program na przykład do postaci

```
10 INPUT "NACHYLENIE="; K
```

```
20 CLS: Y1 = 0
30 FOR X = 0 TO 127
40 Y2 = INT (K * X)
45 FOR Y = Y1 TO Y2
50 IF Y>47 THEN GOTO 10
55 SET (X, 47 - Y)
60 NEXT Y
65 Y1 = Y2
70 NEXT X
80 GOTO 10
```



Rys. 2.21. Wykres prostej $y = 2x$ na siatce punktów

Przy okazji nowa wersja łagodzi nieco „schody" widoczne przy wykreślaniu odcinków ukośnych. Zauważmy jeszcze jedną pułapkę: wartości k równej 1 powinno odpowiadać nachylenie odcinka do poziomu równe 45°, tymczasem na ekranie jest ono dużo większe. Wynika to z faktu, że ekran monitora nie jest kwadratowy ani w sensie długości boków (3:4), ani w sensie liczby punktów w pionie i w poziomie (48:128). Poprawa efektu wymaga skorygowania obliczonej wartości współrzędnej y, zgodnie ze współczynnikiem

$$\frac{4}{3} \cdot \frac{48}{128} = \frac{1}{2}$$

To znaczy, w poprzednim programie musimy zastąpić instrukcję z linii 40 instrukcją

```
40 Y2 = INT (K * X/2)
```

Pomimo wszystkich poprawek rezultat nie jest rewelacyjny, a co gorsza, jest niemożliwy do poprawienia. Schodkowy kształt odcinka wynika po prostu ze zbyt małej rozdzielczości (zbyt małej liczby elementarnych punktów) obrazu w mikrokomputerze *Meritum*.

Podsumowanie

— Użycie funkcji @ i *TAB* w instrukcji *PRINT* pozwala na łatwe redagowanie wyświetlanych na ekranie wyników;
— znaki graficzne (o kodach 128...191) oraz instrukcje: *SET*, *RESET*, *POINT*, których argumentami są współrzędne punktów ekranu, pozwalają na sporządzanie prostych rysunków;
— możliwości grafiki niskorozdzielczej nie są duże, ze względu na zbyt małą liczbę punktów.

Pytania i zadania

1. Wyjaśnij działanie funkcji @.
2. Jakie znasz sposoby tabulacji tekstu wyświetlanego przez instrukcję *PRINT* ?
3. Czym różnią się instrukcje *SET*, *RESET* od instrukcji *PRINT* ?
4. Dlaczego w programie rysowania odcinka używamy instrukcji *SET (X, 47 − Y)*, a nie *SET (X,Y)*?
5. Potraktuj ekran jako obraz prawej połowy układu współrzędnych. Napisz podprogram rysowania pionowego odcinka od punktu (x, y_1) do punktu (x, y_2) działający dla dowolnych wartości y_1, y_2 (mieszczących się oczywiście na ekranie).

2.7
Grafika mikrokomputera Spectrum

Jak pamiętamy z rozdz. 1, użyteczna część ekranu pozostawiona do dyspozycji programu jest podzielona na 22 linie po 32 znaki każda. Linie są numerowane z góry na dół od 0 do 21, kolumny znaków są numerowane z lewa na prawo od 0 do 31. Para liczb: numer linii, numer kolumny jednoznacznie określa pozycję znaku na ekranie. Wykorzystuje to instrukcja *PRINT* z **funkcją *AT*** o postaci

PRINT AT linia, kolumna; lista elementów

w której *lista elementów* może być sporządzana w taki sam sposób jak w zwykłej instrukcji *PRINT*, a para argumentów: *linia, kolumna* określa pozycję znakową, od której rozpocznie się wyświetlanie. Poza innym sposobem określania położenia wyświetlanego na ekranie znaku, instrukcja *PRINT AT* działa tak samo jak instrukcja *PRINT* @ w mikrokomputerze *Meritum*. Na przykład program

```
10 FOR i = 1 TO 5
20 FOR j = 1 TO 2 * i-1
30 PRINT AT 8 + i, 16 − i + j; " * "
40 NEXT j
50 NEXT i
```

spowoduje wyświetlenie w środku ekranu trójkąta zbudowanego z gwiazdek. Podobnie jak w mikrokomputerze *Meritum* możemy też stosować w *Spectrum* **funkcję *TAB***, która może być umieszczona wewnątrz instrukcji *PRINT*. Na przykład program

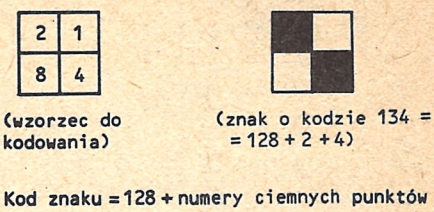
```
10 FOR i = 0 TO 6
20 PRINT TAB 5 * i; INK i; " * ";
30 NEXT i
```

spowoduje wyświetlenie 7 gwiazdek w coraz jaśniejszych kolorach, rozmieszczonych w linii 2 co 5 kolumn. Jeżeli wartość argumentu funkcji *TAB* wskazuje na potrzebę cofnięcia miejsca wydruku, np.

PRINT AT 2,30; "A"; TAB (10); "B"

to następuje automatyczne przejście do następnej linii.

Znaki graficzne dostępne w mikrokomputerze *Spectrum* są umieszczone na klawiaturze nad klawiszami oznaczonymi cyframi 1...8. Każdy znak jest zbudowany z czterech punktów (rys. 2.22), z których każdy może być jasny



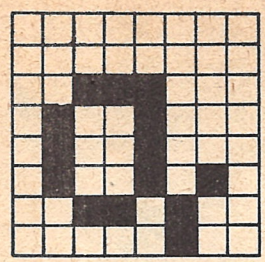
Rys. 2.22. Znak graficzny mikrokomputera *Spectrum*

lub ciemny. 16 znaków graficznych o kodach 128...143 odpowiada wszystkim możliwym kombinacjom punktów jasnych i ciemnych. Kod znaku może być obliczony na podstawie wzorca kodowania pokazanego na rys. 2.22. Z klawiatury znaki graficzne są wprowadzane w trybie graficznym (kursor *G*), przy czym naciśnięcie pojedynczego klawisza 1...8 wprowadza znak, a naciśnięcie klawisza wraz z klawiszem funkcyjnym *CAPS SHIFT* wprowadza jego negatyw. Podobnie jak w mikrokomputerze *Meritum* możliwe jest też posłużenie się funkcją *CHR$*.

Łączna liczba punktów, z których powstają na ekranie znaki graficzne wynosi zaledwie 2816 (32 · 22 · 4), co wystarcza do sporządzania tylko bardzo prostych rysunków. Znacznie większą rozdzielczość zapewnia posługiwanie się elementarnymi punktami ekranu, z których są zbudowane wzorce wszystkich wyświetlanych znaków (por. p. 1.5). W tym zakresie mikrokomputer Spectrum zdecydowanie góruje nad *Meritum*. Zauważmy, że ponieważ wzorzec każdego znaku składa się z 64 punktów elementarnych, łączna liczba punktów na ekranie wynosi aż 45056 (32 · 22 · 64), co pozwala na sporządzanie rysunków o dużej dokładności.

Pierwszą możliwością wykorzystania punktów elementarnych jest **zdefiniowanie własnych znaków graficznych**. Znaki te, o kodach 144...164, mogą być wprowadzane z klawiatury przez naciskanie klawiszy literowych *A*...*U* w trybie graficznym (kursor *G*). Zdefiniowanie znaku zaczynamy od zaprojektowania wzorca znaku. Do dyspozycji mamy kwadrat 64 kropek ułożonych w 8 wierszach po 8 kropek w wierszu (rys. 2.23). Jeżeli ma to być znak tekstowy, to należy przewidzieć obwódkę oddzielającą znak w linii tekstu od znaków sąsiednich (z góry, z dołu i z boków). Po zaprojektowaniu wzorca opisujemy znak w postaci ośmiu liczb dwójkowych, z których każda opisuje jeden wiersz wzorca. Każda liczba składa się z ośmiu cyfr dwójkowych (0 lub 1) przypisanych punktom wzorca w konwencji 0 — punkt w kolorze papieru, 1

— punkt w kolorze atramentu (por. rozdz. 1.5). Na przykład wzorzec litery *ą* z rys. 2.23 możemy opisać



```
00000000
00000000
00111000
01001000
01001000
01001010
00110100
00000100
```

Rys. 2.23. Wzorzec litery *ą*

Teraz decydujemy, która z liter *A*...*U* ma w trybie graficznym oznaczać nowy znak, a następnie wpisujemy do pamięci mikrokomputera osiem wierszy wzorca za pomocą ośmiu instrukcji o budowie

POKE USR "litera" + numer wiersza, BIN wiersz wzorca

w których *"litera"* jest literą odpowiadającą nowemu znakowi, *numer wiersza* jest kolejnym numerem wprowadzanego *wiersza wzorca* nowego znaku. Wiersze numerujemy od 0 do 7. Na przykład, jeżeli nową literę *ą* będziemy oznaczać przez *A*, to zdefiniowanie jej wzorca wymaga wykonania ośmiu instrukcji

```
POKE USR "A", BIN 00000000
POKE USR "A" + 1, BIN 00000000
POKE USR "A" + 2, BIN 00111000
POKE USR "A" + 3, BIN 01001000
```

itd. wszystkie osiem wierszy. Zamiast ośmiu instrukcji *POKE USR...BIN* możemy napisać specjalny program, który zapyta nas o wzorzec:

```
10 INPUT "litera="; a$
20 FOR i = 0 TO 7
30 INPUT ("wiersz"; i; "="); w
40 POKE USR a$ + i, w
50 NEXT i
```

Argumenty umieszczone w nawiasie w instrukcji *INPUT* zostaną wyświetlone na ekranie przed wprowadzeniem wartości zmiennej *w*.

Po kolejnych zapytaniach programu podajemy literę oraz kolejne wiersze wzorca poprzedzone słowem kluczowym *BIN*

```
litera = ? a
wiersz 0 = ? BIN 00000000
wiersz 1 = ? BIN 00000000
wiersz 2 = ?
```

itd. Wyjaśnienie działania instrukcji *POKE* oraz funkcji *USR* i *BIN* poznamy dopiero w rozdz. 4.

Oprócz znaków definiowanych przez użytkownika mikrokomputer *Spectrum* zapewnia możliwość bezpośredniego wykorzystania punktów do tworzenia dowolnych rysunków. W tym celu do języka *Basic* zostały wprowadzone trzy niestandardowe instrukcje: *PLOT*, *DRAW* i *CIRCLE* umożliwiające rysowanie pojedynczych punktów, odcinków i łuków okręgów. Wszystkie wymienione instrukcje traktują ekran jako obraz pierwszej ćwiartki prostokątnego układu współrzędnych $x0y$. Punkt o współrzędnych (0,0) leży w lewym dolnym rogu ekranu, oś $0x$ jest położona wzdłuż dolnego brzegu ekranu w zakresie wartości: $x = 0...255$, natomiast oś $0y$ wzdłuż lewej pionowej krawędzi ekranu w zakresie wartości $y = 0...175$.

Instrukcja rysowania punktu *PLOT*, o postaci

PLOT x,y

stawia kropkę w kolorze atramentu w punkcie ekranu o współrzędnych x,y. Wartości wyrażeń x i y występujących w instrukcji *PLOT* powinny zawierać się w granicach $x = 0...255$, $y = 0...175$. W przeciwnym przypadku wyświetlany jest komunikat błędu:

B Integer ouf of range

co znaczy *wartość całkowita poza dopuszczalnym zakresem*.

Przykładem wykorzystania instrukcji *PLOT* może być program rysowania wykresu funkcji $y = 5\sqrt{x}$

```
10 FOR x = 0 TO 255
20 PLOT x, 5 * SQR (x)
30 NEXT x
```

W taki sam sposób możemy rysować wykresy innych funkcji, np. funkcji trygonometrycznych. Dodatkowym efektem wykonania instrukcji *PLOT x,y* jest ustawienie w punkcie (x,y) znacznika wykorzystywanego przez instrukcję *DRAW*.

Instrukcja rysowania linii *DRAW* może wystąpić w dwóch postaciach. W prostszej

DRAW Δx, Δy

powoduje narysowanie na ekranie odcinka od punktu (x,y), w którym ostatnio znajdował się znacznik do punktu o współrzędnych $(x + \Delta x, y + \Delta y)$. Wartości argumentów Δx, Δy mogą być dodatnie lub ujemne, natomiast współrzędne punktu końcowego powinny zawierać się w zakresie $x + \Delta x = 0...255$, $y + \Delta y = 0...175$. W przeciwnym razie sygnalizowany jest błąd.

Po zakończeniu rysowania znacznik jest przesuwany do punktu końcowego $(x + \Delta x, y + \Delta y)$. Jako przykład wykorzystamy program z p. 2.6 i narysujemy dwie przekątne ekranu dwoma metodami

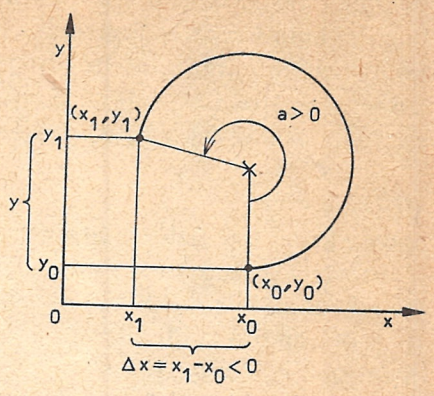
```
10 LET k = 176 / 256
20 FOR x = 0 TO 255
30 PLOT x, k * x
40 NEXT x
50 PLOT 0,175: DRAW 255, -175
```

Instrukcja *PLOT* w linii 50 służy do początkowego ustawienia znacznika. W czasie wykonania programu Czytelnik powinien wzrócić uwagę na dużą różnicę szybkości kreślenia linii. Wyjaśnienie tego faktu podamy w rozdz. 4.

Druga postać instrukcji *DRAW* wymaga dodatkowego argumentu *a*:

DRAW Δx, Δy, *a*

którym jest kąt obrotu. Wykonanie instrukcji powoduje narysowanie łuku okręgu o początku w punkcie (x,y), w którym znajdował się znacznik, końcu w punkcie o współrzędnych $(x + \Delta x, y + \Delta y)$ i o kącie środkowym *a* (rys. 2.24)



Rys. 2.24. Rysowanie łuku okręgu

Znak argumentu *a* wskazuje kierunek obrotu: zgodnie z kierunkiem ruchu wskazówek zegara (−) lub przeciwnie (+). Wartość *a* powinna być podana w radianach. Przypomnijmy, że jeżeli znamy wartość kąta *d* w stopniach, to wartość w radianach wynosi $a = d * \pi / 180$. Po zakończeniu rysowania znacznik jest przesuwany do punktu końcowego $(x + \Delta x, y + \Delta y)$. Jako przykład narysujmy figurę złożoną z dwóch połówek okręgu, przypominającą sinusoidę

```
10 PLOT 0,100
20 DRAW 50, 0, -PI
30 DRAW 50, 0, PI
```

Ostatnią instrukcją graficzną jest **instrukcja *CIRCLE*** powodująca narysowanie okręgu o zadanym środku i promieniu

CIRCLE x, y, r

Argumenty *x*, *y* określają współrzędne środka, *r* oznacza długość promienia. Używając instrukcji *CIRCLE* i *DRAW* narysujemy na ekranie tarczę strzelecką

```
10 PLOT 45, 83 : DRAW 164,0
20 PLOT 127,1 : DRAW 0,164
30 FOR r = 8 TO 80 STEP 8
40 CIRCLE 127,83, r : PRINT AT 10,15-r/8; 10-r/8
50 NEXT r
```

Wewnątrz wszystkich instrukcji graficznych dopuszczalne jest użycie wszystkich funkcji modyfikujących, używanych wewnątrz instrukcji *PRINT*, tzn. *INK, PAPER, FLASH, BRIGHT, INVERSE, OVER*. Należy jednak pamiętać o istniejących różnicach:

— funkcje zmieniające atrybuty (*INK*, *PAPER*, *FLASH* i *BRIGHT*) działają zawsze w obrębie całej pozycji znakowej (8 x 8 punktów);
— funkcje *INVERSE* i *OVER* dotyczą tylko rysowanych punktów, przy

czym np. *PLOT INVERSE 1; x,y* stawia kropkę w kolorze papieru (wymazuje punkt), a *PLOT OVER 1; x, y* zmienia kolor kropki na przeciwny (z atramentu na papier lub odwrotnie).

Przy sporządzaniu bardziej skomplikowanych rysunków pożyteczna jest możliwość sprawdzenia stanu punktu. Realizuje to **funkcja badania stanu** *POINT (x, y)*
której wartością jest 0, jeżeli punkt jest w kolorze papieru, i 1, jeżeli ma kolor atramentu. Np.

```
10 PLOT 100, 100 : PRINT POINT (100, 100)
20 PLOT OVER 1; 100, 100 : PRINT POINT (100, 100)
```

Podsumowanie

— mikrokomputer *Spectrum* dysponuje grafiką niskorozdzielczą obejmującą: funkcje tabulacyjne *AT* i *TAB* oraz znaki graficzne (o kodach *128...143*);
— istnieje możliwość definiowania własnych znaków graficznych (na siatce punktów 8 × 8);
— instrukcje graficzne: *PLOT*, *DRAW*, *CIRCLE* i funkcja *POINT* umożliwiają rysowanie w grafice wysokorozdzielczej;
— możliwość używania kolorów jest w grafice wysokorozdzielczej ograniczona, funkcje *INVERSE* i *OVER* umożliwiają wymazywanie i nakładanie rysunków.

Pytania i zadania

1. Wyjaśnij działanie funkcji *AT*.
2. Jaki będzie efekt wykonania programu

```
10 PLOT 0,0; DRAW 100,100
20 PLOT OVER 1; 0,0 : DRAW OVER 1; 100,100
```

3. Napisz program powodujący narysowanie na środku ekranu trójkąta równoramiennego o podstawie a i wysokości h.
4. Napisz program powodujący narysowanie na ekranie koła (nie okręgu, tylko „pełnego" koła).

3
Przykłady programów

W dwóch poprzednich rozdziałach zajmowaliśmy się przede wszystkim sposobem programowania mikrokomputera. Obecnie spróbujemy wykorzystać zdobyte wiadomości i potraktować mikrokomputer jako narzędzie, które może nam pomóc w nauce.

3.1
Konstrukcje i wykresy funkcji

Najbardziej naturalnym obszarem zastosowania mikrokomputera w szkole są lekcje matematyki. Wynika to z dwóch powodów:
— matematyka jest przedmiotem, na którym w sposób ścisły formułowane są zarówno problemy jak i metody ich rozwiązywania;
— język matematyki: liczby, formuły (wzory) obliczeniowe i wykresy są najbardziej zbliżone do języka komputerów.

Jednym z przykładów wykorzystania mikrokomputera na lekcji matematyki może być **ilustracja wykresów funkcji połączona z wyjaśnieniem ich właściwości.** Ręczne sporządzanie wykresów funkcji np. trygonometrycznych na tablicy lub na papierze jest pracochłonne i mało dokładne, narysowanie ich na ekranie monitora jest szybsze i znacznie dokładniejsze. Napiszmy program konstruowania wykresu sinusoidy (funkcji $y = \sin x$) na podstawie definicji. Program napiszemy w języku *Basic* — wersja **dla mikrokomputera Meritum.**

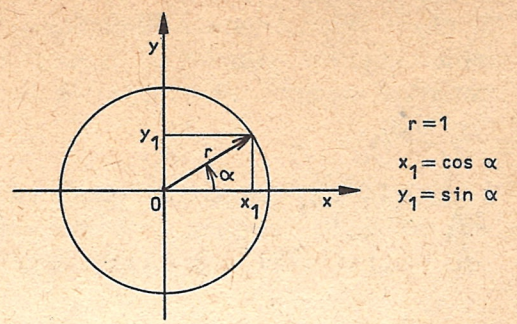
Funkcje trygonometryczne sinus i kosinus określają współrzędne wektora obracającego się względem punktu (0,0) w prostokątnym układzie współrzędnych, w zależności od kąta obrotu wektora od położenia początkowego wzdłuż osi $0x$. Koniec obracającego się wektora wyznacza na płaszczyźnie okrąg (rys. 3.1). Jeżeli przyjmiemy, że wektor ma długość 1 (jego koniec opisuje okrąg o promieniu jednostkowym), to współrzędna x wzdłuż osi $0x$ jest związana z kątem obrotu α zależnością

$$x_1 = \cos \alpha$$

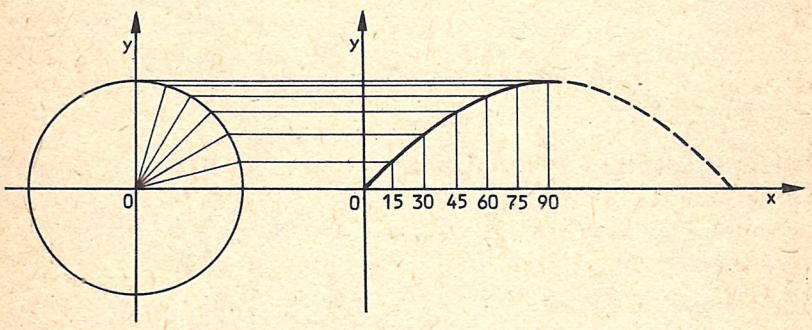
a współrzędna y wzdłuż osi $0y$ zależnością

$$y_1 = \sin \alpha$$

Podane zależności możemy traktować jako definicję funkcji sinus i kosinus, których dziedziną jest kąt obrotu od minus do plus nieskończoności, a przeciwdziedziną wartości współrzędnych od –1 do +1. Przypomnijmy, że kąt obrotu przeciwny do kierunku ruchu wskazówek zegara uważamy za dodatni, a zgodny z kierunkiem ruchu wskazówek zegara za ujemny. Geometryczna konstrukcja sinusoidy polega na narysowaniu kilku promieni okręgu, a następnie na przeniesieniu współrzędnych: kąt nachylenia promienia do osi $0x$ i wartość współrzędnej końca promienia na osi $0y$ na wykres w prostokątnym układzie współrzędnych (rys. 3.2).



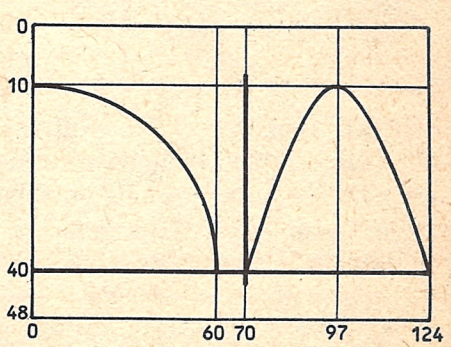
Rys. 3.1. Definicje funkcji sinus i kosinus



Rys. 3.2. Geometryczna konstrukcja sinusoidy

Zaczniemy od rozplanowania powierzchni ekranu. Do dyspozycji mamy prostokąt 48 na 128 punktów, numerowanych z góry na dół i od strony lewej do prawej. Aby oddać kształt okręgu musimy uwzględnić, że szerokość punktu na ekranie monitora jest dwukrotnie mniejsza od jego wysokości (w mikrokomputerze Meritum). Jeżeli ograniczymy rysunek do ćwiartki okręgu, wewnątrz którego poprowadzimy dziesięć promieni (co 10°) oraz do półokresu sinusoidy, to rozmieszczenie rysunku w poziomie może być następujące:

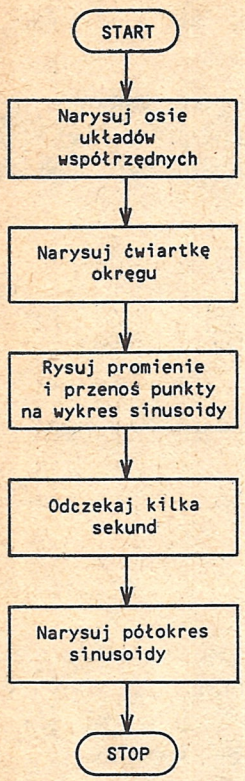
— ćwiartka okręgu — 60 punktów (odpowiada to 30 punktom w pionie),
— półokres sinusoidy (180°) — 54 punkty (po 3 punkty na 10°).



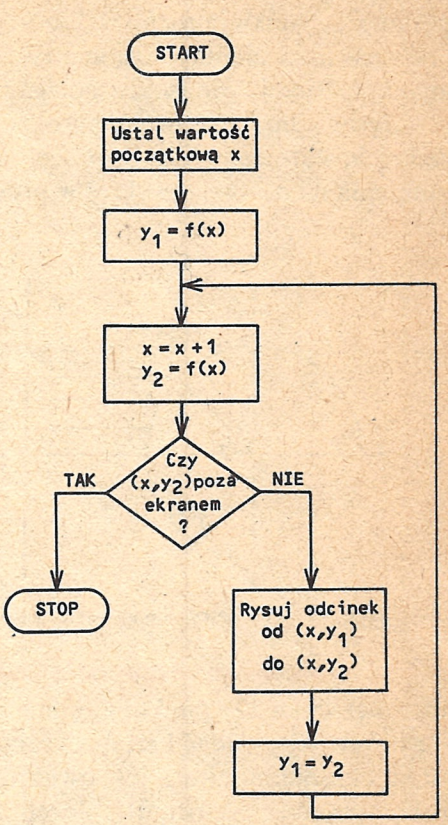
Rys. 3.3. Rozplanowanie powierzchni ekranu

W pionie rysunek zajmie 30 punktów. Aby umieścić rysunek w środkowym pasie ekranu narysujemy oś $0x$ na poziomie wiersza 40, oś $0y$ dla rysunku ćwiartki okręgu w kolumnie 0 i oś $0y$ dla rysunku sinusoidy w kolumnie 70 (rys. 3.3).

Algorytm opracujemy w postaci sieci działań (rys. 3.4) składającej się z sekwencji bloków rysujących kolejne elementy konstrukcji. Tak jak w p. 2.6 wykresy funkcji będziemy rysować najprostszą metodą, obliczając wartość współrzędnej y dla zmieniającej się co 1 wartości współrzędnej x (rys. 3.5).



Rys. 3.4. Sieć działań programu ilustrującego konstrukcję sinusoidy



Rys. 3.5. Generacja wykresu funkcji f(x)

W chwili skokowej zmiany wartości współrzędnej y konieczne jest wypełnienie powstałej luki pionowym odcinkiem (rys. 2.21). W tym celu wykorzystamy podprogram rysowania odcinka pionowego napisany w p. 2.6 (pyt. 5); z rozplanowania rysunku wynika, że wartość $w_0 = 40$.

Dla ułatwienia procesu uruchamiania programu każdy blok sieci działań napiszemy w postaci odrębnego podprogramu.

Pierwszy podprogram — rysowanie osi układów współrzędnych składa się z dwóch pętli: rysowania osi poziomej $0x$ i rysowania obydwu osi pionowych.

```
1100 REM RYSOWANIE OSI
1110 FOR X = 0 TO 127
1120 SET (X,WO)
1130 NEXT X
1140 FOR Y = 5 TO 43
1150 SET (0,Y): SET (70,Y)
1160 NEXT Y
1170 RETURN
```

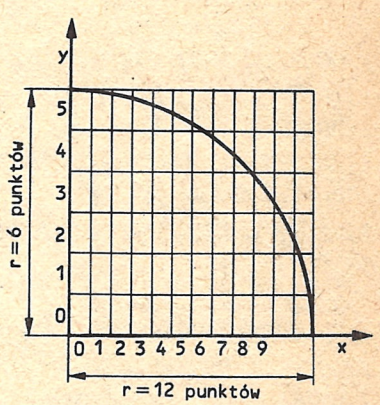
W drugim podprogramie — rysowania ćwiartki okręgu o promieniu r, skorzystamy z równania okręgu

$$x^2 + y^2 = r^2$$

przekształconego do postaci

$$y = \sqrt{r^2 - x^2}$$

Przy obliczaniu wartości współrzędnej y musimy uwzględnić współczynnik korekcyjny o wartości 1/2, wynikający z prostokątnego kształtu punktu ekranu (por. p. 2.6). Dla zachowania symetrycznego kształtu okręgu długość promienia mierzona w punktach wzdłuż osi $0y$ musi być dwukrotnie mniejsza od długości promienia mierzonej w punktach wzdłuż osi $0x$ (rys. 3.6).



Rys. 3.6. Okrąg o promieniu r na siatce punktów ekranu

```
1200 REM RYSOWANIE OKREGU
1210 R2 = R ↑ 2
1220 Y1 = R/2
1230 FOR X = 0 TO 60
1240 Y2 = (SQR (R2 - X * X)) / 2
1250 GOSUB 1000 : REM PIONOWY ODCINEK
1260 Y1 = Y2
1270 NEXT X
1280 RETURN
```

Podprogram realizuje algorytm opisany siecią działań z rys. 3.5. Wartość początkowa $y_1 = r$ (instrukcja *1220 Y1 = R / 2*) odpowiada wartości początkowej $x = 0$.

Najbardziej skomplikowanym podprogramem jest podprogram rysowania promieni i przenoszenia punktów na wykres sinusoidy. Promienie będziemy kreślić jako odcinki o nachyleniu wzrastającym co 10°, prowadzone od środka aż do przecięcia z linią okręgu. Warunek zakończenia rysowania promienia ma postać nierówności

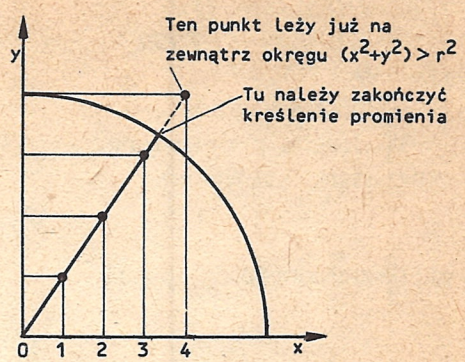
$$x^2 + y^2 \geq r^2$$

w której musimy jeszcze uwzględnić współczynnik korekcyjny 1 / 2

W rezultacie otrzymujemy warunek

$$y \geqslant \frac{1}{2}\sqrt{r^2 - x^2}$$

który musi być sprawdzany nie po obliczeniu kolejnej wartości $y_2(x)$ jak na rys. 3.5, ale w czasie rysowania każdego odcinka od (x, y_1) do (x, y_2). W przeciwnym razie promień mógłby okazać się zbyt krótki (rys. 3.7).



Rys. 3.7. Rysowanie promienia okręgu

```
1300 REM KONSTRUKCJA PUNKTOW SINUSOIDY
1310 B = 10 * 3.14159/180
1320 FOR I = 1 TO 9
1330 IF I = 9 THEN GOTO 1500
1340 K = TAN (B * I)
1345 REM PROMIEN O NACHYLENIU K
1350 Y1 = 0
1360 FOR X = 0 TO 70
1370 RX = (SQR (R2 - X * X)) / 2
1380 Y2 = INT (K * X / 2)
1390 FOR Y = Y1 TO Y2
1400 SET (X,W0 - Y)
1410 IF Y > = RX THEN GOTO 1510
1420 NEXT Y
1430 Y1 = Y2
1440 NEXT X
1500 X = 0 : Y = 30
1505 REM ODCINEK POZIOMY
1510 X2 = 70 + 3 * I
1520 FOR Z = X TO X2
1530 SET (Z,W0 - Y)
1540 NEXT Z
1545 REM ODCINEK PIONOWY
1550 X = X2 : Y1 = 0 : Y2 = Y
1560 GOSUB 1000
1570 NEXT I
1580 RETURN
```

Wartość *B* obliczona w linii programu 1310 to po prostu kąt 10° wyrażony

w radianach. Specjalne potraktowanie wartości $I = 9$ (linie programu 1330, 1500) wynika z faktu, że tangens 90° jest nieokreślony i instrukcja w linii 1340 n e mogłaby być wykonana. Zmienna *R2* ma wartość r^2 obliczoną w poprzedn m podprogramie.

Odczekanie kilku sekund dla obejrzenia konstrukcji najłatwiej zrealizować przez wielokrotne obliczenie przez mikrokomputer wartości jakiejś funkc i

```
1500 REM ODCZEKANIE
1510 FOR X = 0 TO 100
1520 Y = SQR (X)
1530 NEXT X
1540 RETURN
```

Frzy okazji możemy ocenić, ile czasu zajmuje mikrokomputerowi obliczenie pierwiastka. Dla sprawdzenia poprawności naszej konstrukcji napiszemy (posługując się znów algorytmem z rys. 3.5) podprogram rysujący połowę sinusoidy

```
1700 REM RYSOWANIE SINUSOIDY
1710 Y1 = 0
1720 FOR Z =0 TO 54
1730 Y2 = (R * SIN (B/3 * Z))/2
1740 X = 70 + Z
1750 GOSUB 1000
1760 Y1 = Y2
1770 NEXT Z
1780 RETURN
```

Dysponując tyloma podprogramami z łatwością napiszemy program głów-ny konstruujący sinusoidę, zgodnie z algorytmem opisanym przez sieć działań z rys. 3.4

```
10 REM KONSTRUKCJA SINUSOIDY
20 CLS
30 R = 60 : W0 = 40
40 GOSUB 1100
50 GOSUB 1200
60 GOSUB 1300
70 GOSUB 1600
80 GOSUB 1700
90 GOTO 90
```

Aby nie dopuścić do zniszczenia rysunku przez wyświetlenie komunikatu *READY* program zakończyliśmy instrukcją skoku do samej siebie. W ten sposób powstała niekończąca się pętla, w czasie której obszar na ekranie pozostaje niezmieniony. Przerwanie (zakończenie) wykonania programu wymaga naciśnięcia klawisza *BREAK*.

Użytkownicy mikrokomputera *Meritum*, którzy napiszą i uruchomią program, będą mogli przy okazji ocenić ograniczone możliwości grafiki tego

komputera. Przeniesienie naszego programu na mikrokomputer *Spectrum* jest możliwe, wymaga jednak wprowadzenia dość znacznych zmian związanych z innym sposobem określania położenia wyświetlanych na ekranie punktów.

Zadania

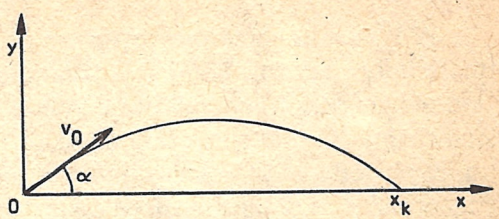
1. Napisz program rysujący wykres funkcji sinus w taki sposób, aby zmieścić na ekranie dwa okresy sinusoidy (bez konstrukcji geometrycznej).
   Potraktuj ekran jako obraz prawej połowy układu współrzędnych (rys. 5.5) z osią $0x$ na poziomie $w_0 = 24$. Skalę $0x$ możesz przyjąć jako 6° na 1 punkt, wysokość sinusoidy na osi $0y$ 20 punktów.
2. Napisz program rysujący wykres funkcji kosinus.
3. Napisz program konstrukcji sinusoidy na mikrokomputer *Spectrum*. Masz do wyboru dwie drogi: albo przekształcić program napisany dla *Meritum*, albo napisać nowy program, korzystając tylko z części opracowanych sieci działań.
   a) Przekształcenie programu napisanego dla mikrokomputera *Meritum*
   Punkt ekranu mikrokomputera *Spectrum* jest kwadratowy, musimy więc usunąć ze wszystkich podprogramów korekcyjne dzielenie wartości współrzędnej y przez 2. Oś $0x$ musi zostać przesunięta np. do linii 30, co wymaga zmiany w podprogramie: „Rysowanie osi". Numeracja linii ekranu wzrasta z dołu do góry, zgodnie z kierunkiem osi $0y$. Uwzględniając przesunięcie osi $0x$ musimy w podprogramach rysowania odcinka, okręgu i sinusoidy zastąpić instrukcję *SET (X,40 − Y)* instrukcją *PLOT X, 30 + Y*.
   Dodatkowo musimy uwzględnić drobne różnice implementacyjne, jak np. konieczność użycia słowa *LET*. Po uruchomieniu programu możesz poprawić proporcje rysunku: zwiększyć promień R (wymaga to również odsunięcia sinusoidy poza kolumnę 70) i poszerzyć sinusoidę, przyjmując nie 3, lecz np. 6 punktów na 10°.
   b) Napisanie nowego programu
   Możemy zrezygnować z podprogramu rysowania ćwiartki okręgu i podprogramu rysowania pionowego odcinka i wykorzystać w ich miejsce instrukcję *DRAW*. Pociągnie to za sobą zmianę sposobu rysowania promieni — współrzędne punktów końcowych możemy obliczyć jako: $x_2 = r \cos a$, $y_2 = r \sin a$ i rysować promienie za pomocą instrukcji *DRAW*, dla kolejnych, zmieniających się co 10° wartości kąta a.

3.2
Komputerowy eksperyment

Przeanalizujemy za pomocą komputera problem rzutu ukośnego lub, jak kto woli, strzału, pod zadanym kątem i z zadaną prędkością początkową, czyli tor punktu w rzucie ukośnym (rys. 3.8). Równanie opisujące badane zjawisko pamiętamy z lekcji fizyki. Rozkładamy prędkość początkową na składowe działające w kierunkach osi $0x$ i $0y$

$$v_x = v_0 \cos \alpha$$
$$v_y = v_0 \sin \alpha$$

po czym zauważamy, że ruch w pionie (wzdłuż osi $0y$) jest ruchem jednostajnie opóźnionym, a ruch w poziomie (wzdłuż osi $0x$) jest ruchem jednostajnym.



Rys. 3.8. Rzut ukośny

Stąd otrzymujemy równania

$$x = v_x t = v_0 t \cos\alpha$$
$$y = v_y t - \frac{1}{2} gt^2 = t(v_0 \sin\alpha - \frac{1}{2} gt)$$

w których t oznacza czas, a g przyspieszenie ziemskie.

Uzyskane równania stanowią model matematyczny opisujący rzut ukośny.

Aby wyznaczyć zasięg rzutu wystarczy wyznaczyć punkt, w którym pocisk zetknie się z ziemią, czyli punkt o współrzędnej $y = 0$. Warunek ten jest spełniony, gdy

$$t(v_0 \sin\alpha - \frac{1}{2} gt) = 0$$

co ma miejsce w dwóch przypadkach: albo gdy $t = 0$ (na początku lotu), albo gdy $v_0 \sin\alpha = \frac{1}{2} gt$ (na końcu lotu). A więc lot pocisku kończy się w chwili

$$t_k = \frac{2v_0 \sin\alpha}{g}$$

Podstawiając obliczoną wartość t_k do wzoru na x, otrzymujemy

$$x_k = v_0 \cos\alpha \cdot \frac{2v_0 \sin\alpha}{g} = \frac{2v_0^2 \sin\alpha \cos\alpha}{g} = \frac{v_0^2 \sin 2\alpha}{g}$$

Zasięg rzutu wzrasta jednostajnie z kwadratem prędkości początkowej v_0, natomiast przy stałej prędkości v_0 osiąga maksimum dla wartości $\alpha = 45°$ ($\sin 2\alpha$ przyjmuje tu swoje maksimum równe 1).

Dyskusja wpływu parametrów eksperymentu (rzutu): prędkości początkowej v_0 i kąta α na wartość x_k są przykładem analitycznego rozwiązania pro-

blemu. Mikrokomputer może tu być przydatny do obliczenia końcowych wzorów, czyli do obliczenia zasięgu rzutu dla zadanych parametrów v_0 i α. Realizuje to program

```
10 INPUT "V0="; V0
20 INPUT "A="; A
30 LET A = A * 3.14159/180
40 PRINT "X="; V0 * V0 * SIN (2 * A)/9.81
```

Znajomość równań opisujących tor ruchu punktu daje jednak znacznie większe możliwości. **Równania stanowią matematyczny model badanego zjawiska i mogą być wykorzystane do przedstawienia przebiegu zjawiska na ekranie.** W tym celu napiszemy program obliczający kolejne położenia punktu w miarę upływu czasu. Ponieważ działanie programu będzie dość powolne, uzyskamy w efekcie ruchomy obraz punktu pozostawiającego za sobą ślad toru. Zauważmy, że taki sam efekt uzyskamy filmując np. lot rzeczywistego pocisku i wyświetlając film w zwolnionym tempie. Wykonanie programu możemy więc uważać za przeprowadzenie komputerowego eksperymentu zastępującego trudny do przeprowadzenia eksperyment fizyczny. Uzyskanie tych samych rezultatów, tzn. zbadanie kształtu toru w zależności od wartości parametrów początkowych: kąta i prędkości wyrzutu, w realnym doświadczeniu wymaga specjalnego sprzętu i nastręcza wiele trudności technicznych.

Dokładność komputerowego eksperymentu, tzn. wierność w stosunku do rzeczywistości, zależy przede wszystkim od dokładności modelu. Na przykład nasz model nie uwzględnia wpływu oporu powietrza, jest więc właściwie modelem rzutu ukośnego w próżni. Jeżeli chcemy go jednak uważać za model rzutu w powietrzu, to musimy pamiętać, że wyniki będą obarczone pewnym błędem, a obliczony tor pocisku będzie nieco różny od rzeczywistej krzywej balistycznej. Ocena błędu jest trudna — błąd nie jest stały i zależy od warunków atmosferycznych panujących w czasie rzeczywistego eksperymentu.

Program obliczający tor pocisku napiszemy w wersji języka Basic dla mikrokomputera Spectrum. Zaadaptowanie programu do postaci akceptowanej przez *Meritum* wymaga nieco wysiłku, a osiągnięte rezultaty będą znacznie gorsze ze względu na mizerne możliwości graficzne tego mikrokomputera.

Podobnie jak poprzednio wykres będziemy rysować obliczając wartości współrzędnej y toru pocisku przy zmieniającej się co 1 wartości współrzędnej x. W tym celu przekształcimy równanie toru pocisku tak, aby uzyskać zależność $y(x)$. Obliczamy z pierwszego równania $t = \frac{x}{v_0 \cos\alpha}$ i podstawiamy do drugiego

$$y = \frac{x}{v_0 \cos\alpha}\left(v_0 \sin\alpha - \frac{gx}{2v_0 \cos\alpha}\right)$$

Ponieważ wartość przyspieszenia ziemskiego $g = 9{,}81$, więc ostatecznie otrzymujemy zależność

$$y = \frac{x}{v_0 \cos\alpha}\left(v_0 \sin\alpha - 4.905\ \frac{x}{v_0 \cos\alpha}\right)$$

Przy pisaniu programu posłużymy się nieco zmodyfikowanym algorytmem z rys. 3.5. Do rysowania odcinka łączącego ostatni narysowany punkt z punktem nowo obliczonym (wewnętrzny blok pętli sieci działań) wykorzystamy instrukcję *DRAW*. Program rysowania toru pocisku dla zadanych wartości kąta α oznaczonego w programie *a* i prędkości początkowej v_0 ma teraz postać

```
10 INPUT "v0=";v0
20 INPUT "a=";a
30 LET a=PI/180*a
40 LET y1=0 : PLOT 0,0
50 FOR x=1 TO 255
60 LET x1=x: (v0 * COS a)
70 LET y2 = x1 * (v0 * SIN a - 4.905 * x1)
80 IF y2 < 0 THEN LET y2 = 0
90 LET d=INT (y2-y1) : DRAW 1,d
100 IF y2 < = 0 THEN GOTO 150
110 LET y1 = y1 + d
120 NEXT x
130 INPUT "Strzal za dlugi"; z$ : STOP
150 INPUT ("Zasieg x=";x); z$
```

Zauważmy, że instrukcja napisana w linii 110:

```
110 LET y1=y1+d
```

odpowiada blokowi *y1 = y2* w sieci działań. Konieczność użycia takiej instrukcji wynika ze sposobu zaokrąglania argumentów instrukcji *DRAW*. Jeżeli argumenty te nie są całkowite, to przed rozpoczęciem rysowania ich wartości są sprowadzane do wartości całkowitych przez odrzucenie części ułamkowej, tak samo jak w funkcji *INT*, gdzie *INT* (1.1) = 1, podobnie jak *INT* (1.9 = 1). Ten sposób zaokrąglania może prowadzić do istotnego błędu — 1.9 to już prawie 2, a nie 1. Napisanie programu dokładnie odwzorowującego sieć działań, tzn. zawierającego instrukcje

```
90 DRAW 1,y2-y1
110 LET y1=y2
```

prowadzi do sytuacji, w której inna (zaokrąglona) wartość współrzędnej jest używana do rysowania w instrukcji *DRAW*, a inna jest pamiętana jako wartość zmiennej *y1*. Efekty są dość nieoczekiwane. Po uruchomieniu programu otrzymamy na ekranie wykres toru pocisku i dodatkowo zasięg rzutu. Aby nie dopuścić do zniszczenia rysunku, do wyświetlenia zasięgu użyto instrukcji *INPUT*, która powoduje wyświetlenie żądanej wartości w dolnej części ekranu przeznaczonej na konwersację. Zmienna *x* umieszczona po średniku w instrukcji *INPUT* zostanie potraktowana jako wartość do wyświetlenia, a nie do wprowadzenia, dzięki temu, że została umieszczona wewnątrz nawiasu. Wykonanie programu kończy się po naciśnięciu klawisza *ENTER*.

Zanim przystąpimy do eksploatacji programu, musimy ustalić dopuszczalny zakres parametrów wejściowych *a* i *v* oraz sposób interpretacji wyników (co to znaczy Zasięg *x* = 5? 5 metrów, centymetrów, czy może łokci?). Jeżeli

prędkość będziemy podawać w metrach na sekundę, to zasięg rzutu x będzie obliczony w metrach. Ponieważ w treści programu nie stosowaliśmy żadnych współczynników skalujących, więc wartości współrzędnych x i y są również wyrażone w metrach. Jeden punkt ekranu (zarówno w poziomie jak i w pionie) odpowiada odległości 1 m. Ograniczenie wartości parametrów wynika z wymagania, aby cały wykres toru zmieścił się na ekranie oraz aby wartość funkcji tangens (linia 60) nie przekroczyła dopuszczalnego zakresu. Ograniczenia te najprościej ustalić doświadczalnie

$a < 90$ [°]

$v \leqslant 50$ [m/s]

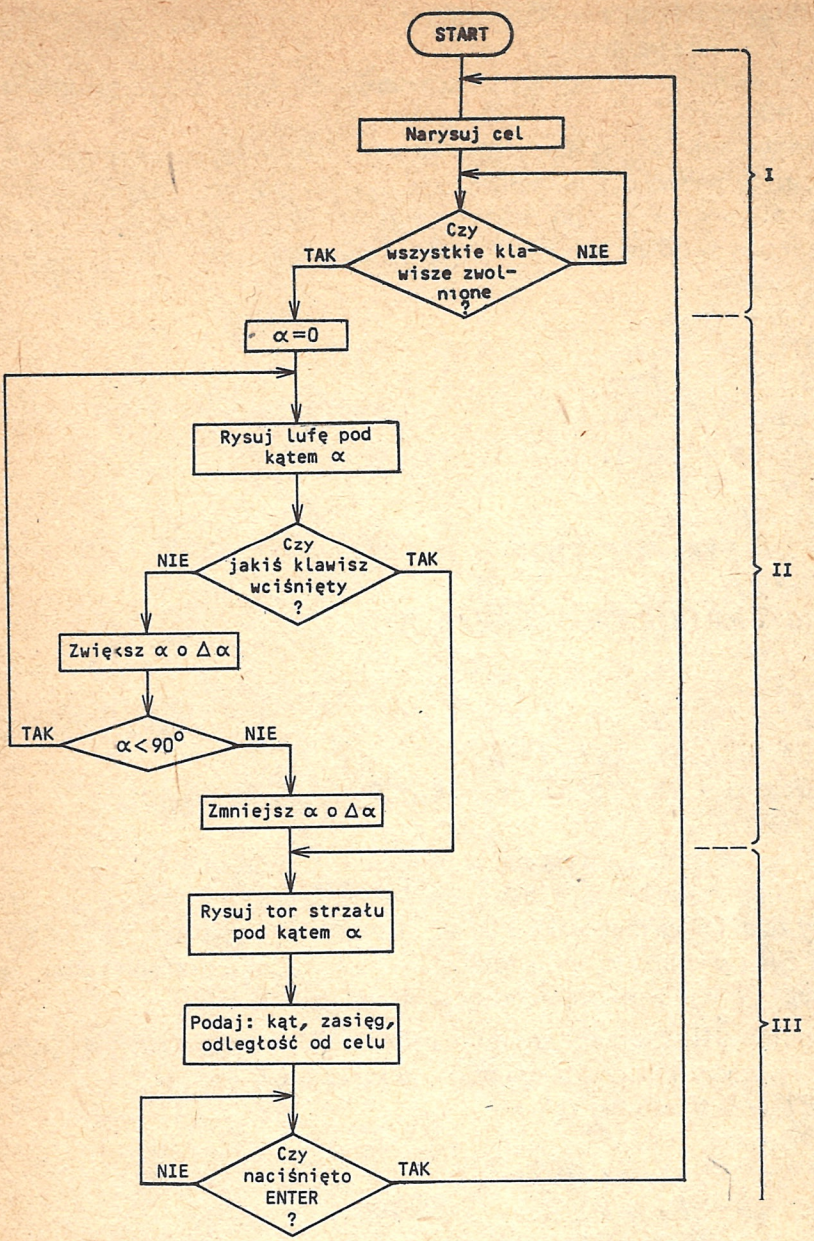
Jeżeli wartość prędkości uznamy za zbyt małą, musimy przeskalować wykres tak, aby jeden punkt ekranu odpowiadał większej odległości, np. 10 m.

Symulacja zjawisk i sytuacji jest również podstawą programowania gier komputerowych. Aby urozmaicić nieco lekcję fizyki, przerobimy nasz program do postaci gry, polegającej na strzelaniu do celu. Sposób grania będzie prosty: w prawym dolnym rogu ekranu pojawi się cel, w lewym lufa podnosząca się powoli od kąta 0° do prawie 90°. W dowolnej chwili możemy oddać strzał, naciskając jakiś klawisz. W tej chwili mikrokomputer rozpocznie obliczanie i rysowanie toru pocisku przyjmując pewną ustaloną z góry wartość prędkości początkowej i wartość kąta równą aktualnemu podniesieniu lufy w chwili strzału. Sieć działań programu pokazano na rys. 3.9. Jak widzimy, samoczynne oddanie strzału nastąpi w chwili, gdy lufa armaty osiągnie maksymalny kąt podniesienia.

Dla ułatwienia program napiszemy w postaci trzech sekcji zaznaczonych na rys. 3.9. Pewnego wyjaśnienia wymaga tekst, „Czy wszystkie klawisze zwolnione", umieszczony w I sekcji programu. W II sekcji programu musimy wykonywać dwie czynności równocześnie: rysować podnoszącą się lufę armaty i sprawdzać, czy został naciśnięty jakiś klawisz. Do sprawdzenia musimy użyć funkcji *INKEY$*, gdyż użycie instrukcji *INPUT* zatrzymałoby wykonanie programu aż do chwili naciśnięcia klawisza *ENTER*. Program wykonuje się tak szybko, że do pierwszego sprawdzenia „czy jakiś klawisz wciśnięty" dojdzie zanim zdążymy zwolnić klawisz kończący polecenie *RUN*, którym uruchomimy nasz program. Stąd konieczne jest wstępne sprawdzenie, czy ten klawisz został już na pewno zwolniony.

Oprócz czynności zaznaczonych w sieci działań, w I sekcji nadamy wartości różnym parametrom, określającym sposób działania programu. Zabieg ten znacznie ułatwia późniejsze modyfikowanie programu

```
10 LET n = 20
20 LET p = 5
30 LET l = 8
40 LET v = 55
50 LET k = PI/(2 * n)
60 PLOT 245,0 : DRAW 10,0
70 PLOT 250,1 : DRAW 0,2
```



Rys. 3.9. Sieć działań programu rysowania toru lotu pocisku

80 IF INKEY$ < >"" THEN GOTO 80

Znaczenie poszczególnych parametrów jest następujące:

n — liczba kolejnych położeń lufy (podczas jej podnoszenia);

p — czas oczekiwania przed podniesieniem lufy do kolejnego, wyższego położenia podany w jednostkach 20 ms;

l — długość lufy (w punktach ekranu);
v — prędkość początkowa;
k — elementarna zmiana kąta nachylenia lufy w trakcie jej podnoszenia (w radianach).

Podnoszenie lufy w II sekcji programu odbywa się przez rysowanie lufy w n kolejnych pozycjach, z nachyleniem określonym przez kąt a (w programie zwiększający się w kolejnych krokach o wartość k). Między kolejnymi położeniami lufy w odstępach ok. 20 milisekundowych p-krotnie sprawdzany jest stan klawiatury

```
100 LET a = 0
110 FOR i = 1 TO n-1
120 LET lx = l * COS a
130 LET ly = l * SIN a
140 PLOT 0,0 : DRAW lx, ly
150 FOR j = 1 TO p
160 PAUSE 1
170 IF INKEY$ < >"" THEN GOTO 250
180 NEXT j
190 PLOT 0,0 : DRAW OVER 1; lx, ly
200 LET a = a + k
210 NEXT i
220 LET a = a - k
250 REM nachylenie strzału = a
```

Umieszczona w linii 160 instrukcja *PAUSE* o ogólnej postaci

PAUSE m

powoduje zatrzymanie wykonania programu na czas $m * 20$ milisekund. Wewnętrzna pętla zawierająca instrukcję *PAUSE* sprawdza w ciągu $p * 20 = 100$ ms, czy został naciśnięty jakiś klawisz. Jeżeli nie, to instrukcja *DRAW* z funkcją *OVER 1* wymazuje obraz lufy z ekranu i program przechodzi do kolejnego obiegu pętli i narysowania następnego położenia lufy.

Ostatnia, III sekcja programu realizuje znany nam już algorytm wykreślania toru pocisku wystrzelonego pod kątem a z prędkością początkową v.

```
300 LET y1 = 0 : PLOT 0,0
310 FOR x = 1 TO 255
320 LET x1 = x / (v * COS a)
330 LET y2 = x1 * (v * SIN a - 4.905 * x1)
340 IF y2<0 THEN LET y2 = 0
350 LET d = INT (y2 - y1) : DRAW 1,d
360 IF y2 = 0 THEN GOTO 400
370 LET y1 = y1 + d
380 NEXT x
390 INPUT ("a=" ; a ;" Pudlo"); t$ : GOTO 410
400 INPUT ("a="; a; " x="; x); t$
410 GOTO 10
```

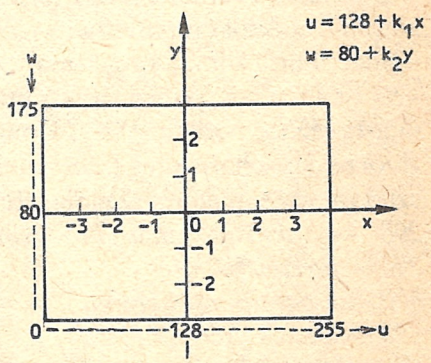
Po zakończeniu rysowania toru program wyświetla wartość kąta strzału i zasięg rzutu. Po naciśnięciu klawisza *ENTER* gra rozpoczyna się od nowa, z tym jednak, że obraz poprzedniego toru pozostaje na ekranie, co pozwala porównać tory pocisku przy różnych kątach nachylenia lufy. Jeżeli chcemy, aby każdy seans gry był odrębną całością, wystarczy zmienić ostatnią linię programu

410 CLS : GOTO 10

i wymazać obraz ekranu przed wznowieniem gry.

Zadania

1. Napisz program rysowania wykresu funkcji sinus (por. p. 3.1, pyt. 1) w taki sposób, aby zmieścić na ekranie trzy okresy sinusoidy ($x = 0...6\pi$).
2. Napisz program rysowania wykresu funkcji e^x w taki sposób, aby umożliwić dowolne skalowanie osi $0x$ i $0y$ (np. tak, aby jednostka odpowiadała 20 punktom na osiach $0x$ i $0y$).
   Ekran musimy potraktować jako obraz I i II ćwiartki układu współrzędnych $x0y$ z osią $0y$ pośrodku. Zależność numeru kolumny u i numeru wiersza w na ekranie od wartości współrzędnych x,y ma postać
   $$u = 128 + k_1 x$$
   $$w = k_2 y$$
   gdzie k_1 i k_2 są współczynnikami skali. Aby wykorzystać dotychczas stosowany algorytm rysowania wykresu funkcji, musimy zmieniać co 1 wartość u. Dla każdej wartości u obliczamy x, dalej y i w. Na wykres nanosimy kolejne punkty o współrzędnych (u, w).
3. Napisz program wykreślania paraboli korzystając z równania o postaci
   $$y - y_0 = a\,(x - x_0)^2$$
   Wartości parametrów a, x_0, y_0 powinny być wprowadzane na początku programu.
   Ekran musimy potraktować jako obraz całego układu współrzędnych (rys. 3.10) o osiach wyskalowanych, zgodnie ze współczynnikami k_1 i k_2 (por. pyt. 2). Wykres możemy rysować używając tej samej metody co w pyt. 2 z tym jednak, że tym razem musimy obliczyć początkową wartość współrzędnej u_1 lewego skrajnego punktu wykresu.



Rys. 3.10. Rysunek do zadania 3: ekran mikrokomputera *Spectrum* jako obraz układu współrzędnych $x0y$

3.3
Rysowanie mapy — komputerowy egzamin

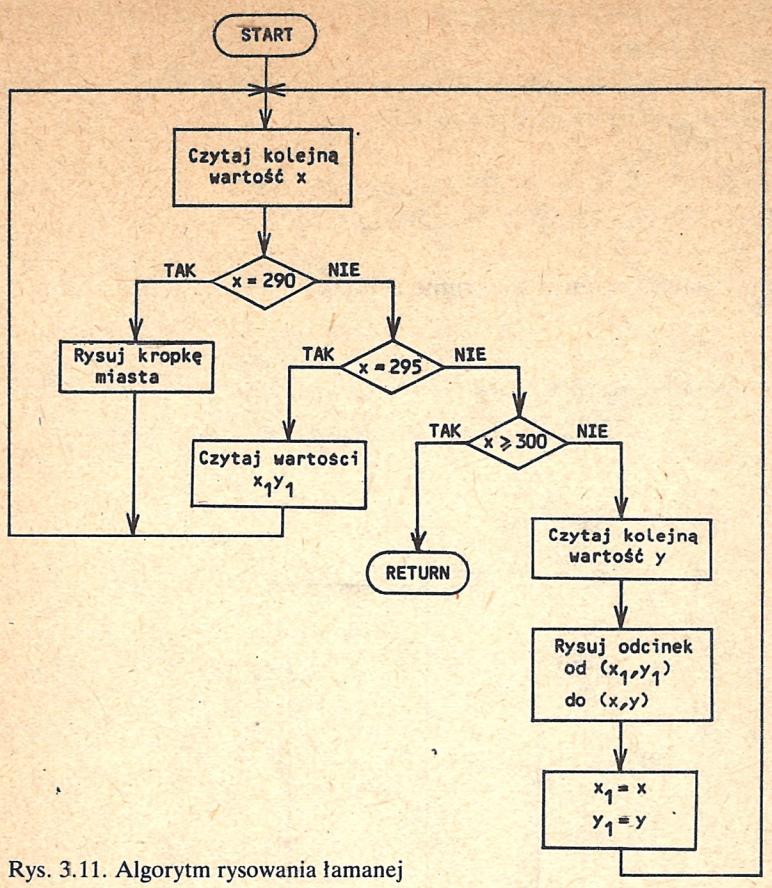
Zgromadzimy w naszym programie dane o prawobrzeżnych dopływach Wisły. Dane te obejmą nie tylko nazwy dopływów, ale także ich przybliżony przebieg na terenie kraju i miejsca wpadania do Wisły. Najbardziej czytelną formą przedstawiania kształtów nieregularnych linii (rzek) i ich wzajemnego położenia jest rysunek. Oczywiście, można te dane zakodować w postaci zespołu liczb — i tak będzie je pamiętał komputer, jednak dla nas byłaby to forma absolutnie nieczytelna. Dlatego polem działania naszego programu będzie ekran, na którym narysujemy fragment mapy przypominający znane mapy konturowe.

Aby spełnić postawione zadania, musimy narysować na ekranie kontur Polski, kształt Wisły, a następnie wszystkie jej dopływy. Zauważmy, że cały rysunek składa się wyłącznie z linii, których rzeczywisty kształt możemy przybliżyć łamaną. Dokładność odwzorowania kształtu będzie tym większa, im więcej punktów załamania uwzględnimy. Przy takiej konstrukcji rysunku zarówno kształt jak i położenie linii mikrokomputer zapamięta w postaci zespołu współrzędnych wszystkich punktów załamania wszystkich linii (rzek). **Ze względu na wymaganą rozdzielczość rysunku, zadanie możemy zaprogramować tylko na mikrokomputerze *Spectrum*;** rozdzielczość ekranu mikrokomputera *Meritum* jest zdecydowanie za mała.

Głównym elementem programu będzie podprogram rysowania łamanej. Ciąg punktów załamania będzie przechowywany w postaci zbioru danych utworzonych za pomocą deklaracji *DATA*. Algorytm rysowania sprowadzimy do odczytywania kolejnych danych ze zbioru i nanoszenia odpowiedniego odcinka na ekranie. Każdy punkt będzie reprezentowany jako para współrzędnych (x, y). Ponieważ dostępne pole ekranu ma wymiary 176 × 256 punktów, więc dopuszczalnym zakresem wartości współrzędnych jest $y = 0...175$, $x = 0...255$. Wartości spoza tego zakresu możemy wykorzystać do zasygnalizowania mikrokomputerowi sytuacji szczególnych, takich jak początek lub koniec linii i koniec programu. Aby lepiej oddać położenie rzek na terenie kraju napiszemy podprogram, który dodatkowo zaznaczy kropkami większe miasta leżące wzdłuż Wisły. Konieczność narysowania miasta zaznaczymy w zbiorze danych również jako sytuację szczególną. W rezultacie wykorzystamy cztery kody sytuacji szczególnych:

290 — rysuj kropkę miasta (o wymiarach 2 × 2 punkty)
295 — początek linii
300 — koniec podprogramu
350 — koniec programu.

Kody 300 i 350 powodują zakończenie podprogramu (*RETURN*). Opisany słownie algorytm możemy przedstawić w postaci sieci działań (rys. 3.11), na podstawie której napiszemy program



Rys. 3.11. Algorytm rysowania łamanej

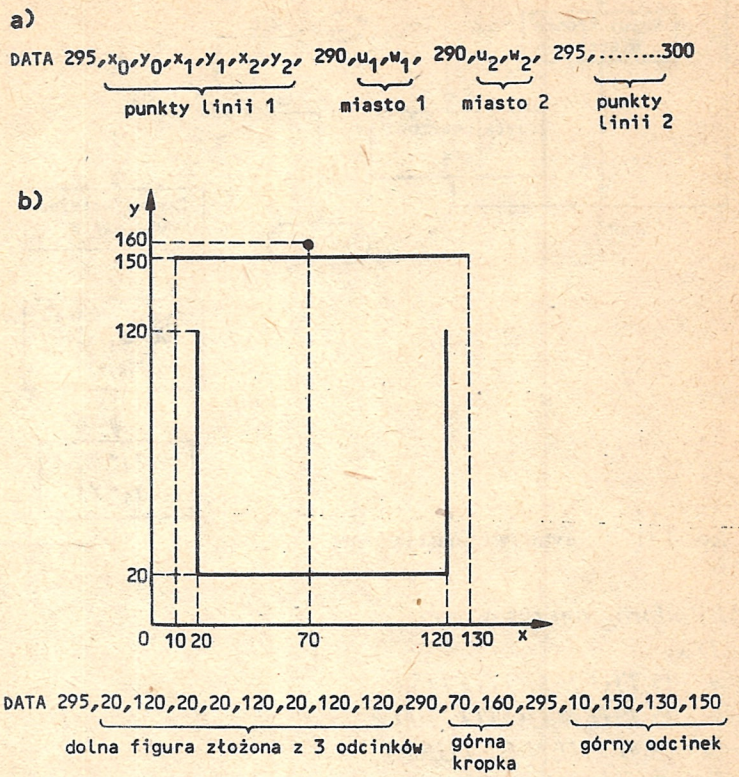
```
1000 REM Podprogram rysowania
1010 READ x
1020 IF x = 290 THEN GOTO 1150
1030 IF x = 295 THEN GOTO 1100
1040 IF x > = 300 THEN RETURN
1049 REM Kolejny punkt zalamania
1050 READ y
1060 DRAW x - x1, y - y1
1070 LET x1 = x : LET y1 = y
1080 GOTO 1010
1099 REM Poczatek rysowania lamanej
1100 READ x1, y1
1110 PLOT x1, y1
1120 GOTO 1010
1149 REM Rysowanie kropki miasta
1150 READ x1, y1
1160 PLOT x1, y1 : PLOT x1 + 1, y1
```

1170 PLOT x1, y1 + 1 : PLOT x1 + 1, y1 + 1
1180 GOTO 1010

Postać danych wymaganych przez podprogram rysowania pokazano na rys. 3.12*a*. Na przykład wywołanie danych

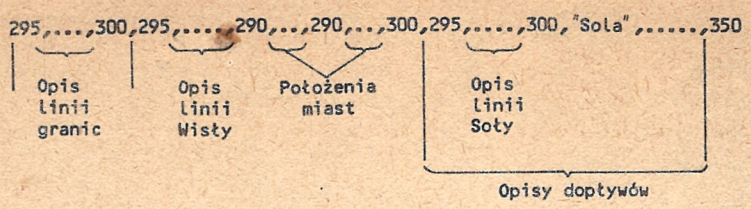
10 GOSUB 1000
20 DATA 295, 20, 120, 20, 20, 120, 20, 120, 120
30 DATA 290, 70, 160, 295, 10, 150, 130, 150, 300
40 STOP

spowoduje narysowanie na ekranie obrazka zamieszczonego na rys. 3.12*b*.



Rys. 3.12. Format danych dla podprogramu rysowania łamanej:
a) format danych; *b)* przykładowy rysunek

Kolejnym elementem programu powinien być opis dopływów w postaci nazw rzek. Możliwości rozmieszczenia nazw na ekranie jest kilka, np. bezpośrednio w pobliżu linii rzek lub w kolumnie obok rysunku. Ciekawa jest jednak możliwość konwersacji z programem, w czasie której program rysuje kolejne dopływy, pytając o ich nazwy. W razie błędnej odpowiedzi program podpowie właściwą nazwę. Nazwy dopływów rozmieścimy w tym samym zbiorze danych co punkty załamania, w postaci tekstów umieszczonych po kodzie 300. Sposób rozmieszczenia danych w zbiorze przedstawiono na rys. 3.13.



Rys. 3.13. Kodowanie danych

```
10 GOSUB 1000 : REM Polska
20 GOSUB 1000 : REM Wisla
30 LET l = 20
35 INPUT "Nacisnij ENTER" ; z$
40 GOSUB 1000 : REM doplyw
50 IF x = 350 THEN GOTO 200
60 READ n$
70 INPUT "Jaki to doplyw?" ; z$
80 IF z$ = n$ THEN GOTO 150
90 INPUT "Nie. Przypomnij sobie?" ; z$
100 IF z$ = n$ THEN GOTO 150
110 INPUT "Pomysl jeszcze raz?" ; z$
120 IF z$ = n$ THEN GOTO 150
130 INPUT AT 0,0 ; ("Nie. To jest "; n$); AT 1,0; "Nacisnij ENTER"; z$
150 PRINT AT l, 240; n$
160 LET l=l - 1 : GOTO 40
200 STOP
```

Funkcja *AT* umieszczona w linii 130 wewnątrz instrukcji *INPUT* działa tak samo jak wewnątrz instrukcji *PRINT*, z tym jednak, że numeracja linii dotyczy dolnej części ekranu przeznaczonej na konwersację użytkownika z mikrokomputerem.

Napisany przez nas program daje trzykrotną szansę odpowiedzi na pytanie. Oczywiście, dobry uczeń odpowie na ogół w pierwszej próbie, dla marnego i trzech prób będzie mało. Ponieważ w każdym przypadku mikrokomputer „wie", za którym razem padła prawidłowa odpowiedź, może więc wystąpić w roli egzaminatora i wystawić ocenę. W tym celu wprowadzimy do programu licznik błędów nastawiony początkowo na wartość 30 (uwzględnione jest 10 dopływów, po 3 próby odpowiedzi na każdy) i zmniejszany o 1 po każdym błędzie. Ocena zostanie wystawiona według skali:

co najmniej 28 punktów	— 5
co najmniej 25	— 4+
co najmniej 23	— 4
co najmniej 20	— 3+
co najmniej 18	— 3
poniżej 18	— 2

Do poprzedniego programu należy dodać instrukcje

```
  5 BORDER 0
 25 LET p = 30
 85 LET p = p-1
105 LET p = p-1
125 LET p = p-1
200 CLS : BORDER 7
210 PRINT AT 8,10; "Ocena =";
220 IF p < 18 THEN PRINT "2"
230 IF p >= 18 THEN PRINT "3"
240 IF p >= 20 THEN PRINT AT 8,16; "3+"
250 IF p >= 23 THEN PRINT AT 8,16; "4"
260 IF p >= 25 THEN PRINT AT 8,16; "4+"
270 IF p >= 28 THEN PRINT AT 8,16; "5";
280 IF p = 30 THEN PRINT AT 9,6; "moje gratulacje"
290 STOP
```

Najbardziej pracochłonnym etapem przygotowania programu omawianego typu, tzn. programu gromadzącego dane nie jest programowanie, lecz utworzenie bazy danych. W naszym przykładzie baza danych jest zbiorem pamiętającym kształt granic Polski i linii rzek. Po przekalkowaniu mapy na papier milimetrowy otrzymujemy zbiór

```
300 REM Granice
310 DATA 295,8,142,10,141,12,141,35,150,41,150,46,157
320 DATA 55,160,76,165,84,159,83,158,80,161,77,163
330 DATA 80,154,84,152,91,152,98,155,120,152,135,152
340 DATA 150,155,153,152,160,148,160,142,161,140,168,122
350 DATA 168,118,169,109,162,103,158,97,158,94,165,91
360 DATA 165,78,167,73,167,69,176,57,175,55,176,47,170,45
370 DATA 153,20,154,10,157,5,153,6,148,9,143,10,140,14
380 DATA 134,16,128,16,121,12,116,15,112,15,108,12,106,9
390 DATA 101,9,101,14,95,18,92,17,91,14,87,14,79,28,71,31
400 DATA 68,30,63,34,65,39,60,38,55,39,50,43,52,37,46,34
410 DATA 38,43,41,46,41,49,36,49,25,54,22,59,18,60,18,56
420 DATA 14,57,19,67,17,73,14,75,12,82,15,90,12,100,13,104
430 DATA 6,113,11,124,10,130,8,142,300
440 REM Wisla i miasta
450 DATA 295,87,19,86,19,85,21,83,27,90,29,91,32,97,31,101,31
460 DATA 105,33,108,33,111,35,116,36,119,39,130,44,137,52,136,65
470 DATA 138,70,130,78,126,84,125,90,116,99,110, 97,104,98,87,105
480 DATA 80,114,72,117,79,126,83,127,82,132,85,135,85,140,83,145
490 DATA 86,152,295,85,149,82,153,295,85,140,92,146,93,152
500 DATA 290,101,32,290,133,50,290,118,98,290,97,102
510 DATA 290,80,114,290,68,117,290,81,151,290,78,155,300
520 REM Sola
```

```
530 DATA 295,87,14,89,17,90,19,92,22,92,32,300,"Sola"
540 REM Skawa
550 DATA 295,101,19,102,20,100,23,96,27,95,32,300,"Skawa"
560 REM Raba
570 DATA 295,105,16,103,17,106,22,104,26,105,27,110,29,110,30
580 DATA 112,30,113,35,300,"Raba"
590 REM Dunajec
600 DATA 295,103,10,102,13,106,16,109,16,113,15,112,18,115,18
610 DATA 118,20,118,26,120,27,121,32,118,38,300,"Dunajec"
620 REM Wisloka
630 DATA 295,124,13,124,15,130,16,130,18,133,18,129,32,131,38
640 DATA 130,40,129,44,300,"Wisloka"
650 REM San
660 DATA 295,155,10,150,12,151,16,145,21,146,23,145,24,145,27
670 DATA 156,27,156,31,144,46,138,51,137,53,300,"San"
680 REM Wieprz
690 DATA 295,164,48,156,50,160,60,148,77,138,76,135,74,300,"Wieprz"
700 REM Narew
710 DATA 295,177,110,174,110,169,115,153,114,150,120,152,123
720 DATA 144,122,140,119,137,122,130,118,127,112,128,109,123,108
730 DATA 122,100,117,100,116,99,300,"Narew"
740 REM Drweca
750 DATA 295,107,129,105,133,103,134,101,134,95,122,86,115
760 DATA 84.115,82,113,300,"Drweca"
770 REM Osa
780 DATA 295,96,134,93,128,90,126,85,127,83,129,300,"Osa",350
```

Ta część pracy jest niewątpliwie najmniej przyjemna.

Zadania

1. Zakoduj w postaci zbioru *DATA* dowolną mapę (kontur kraju lub regionu) i wyświetl ją na ekranie.
2. Rozszerz podprogram rysowania łamanej tak, aby mógł rysować linię złożoną nie tylko z odcinków, ale również łuków. Dodaj do podprogramu możliwość rysowania okręgów.

4
Budowa i programowanie mikrokomputera

Rozdział ten zawiera uzupełniające informacje techniczne dotyczące sposobu działania mikrokomputera. Informacje te nie są niezbędne dla użytkowników programujących w języku Basic. Są one przeznaczone dla Czytelnika interesującego się techniką mikrokomputerową, który chciałby wiedzieć nieco więcej o możliwościach i ograniczeniach swojego mikrokomputera.

4.1
Języki programowania

Programując w języku *Basic* nie zastanawiamy się na ogół, jak są wykonywane poszczególne instrukcje programu. Przyjmujemy po prostu, że stanowią one elementarne i niepodzielne operacje wykonywane przez mikrokomputer. W rzeczywistości tak nie jest. Nawet najprostsze instrukcje języka *Basic* są dla mikrokomputera zbyt skomplikowane do bezpośredniego wykonania. Każda z nich musi zostać najpierw rozłożona na ciąg znacznie prostszych, elementarnych operacji wykonywanych przez układy elektroniczne mikrokomputera. Możemy powiedzieć, że **każdej instrukcji języka *Basic* odpowiada cały podprogram napisany w języku wewnętrznym, bezpośrednio rozumianym przez mikrokomputer.** Wykonanie każdej napisanej przez nas instrukcji języka *Basic* wymaga więc wykonania dwóch oddzielnych kroków:

— przeanalizowania treści instrukcji i przetłumaczenia jej na odpowiedni podprogram w języku wewnętrznym,
— wykonania tego podprogramu.

Tłumaczenie jest wykonywane przez specjalny program tłumaczący, nazywany **translatorem**.

Sposób działania programu tłumaczącego (translatora) może być dwojaki:

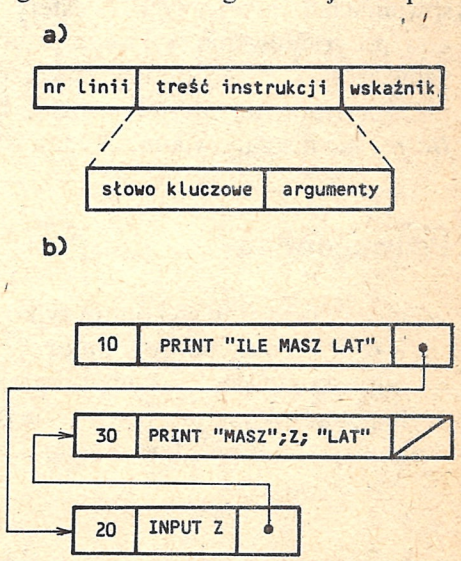
— albo nastąpi jednorazowe przetłumaczenie całego programu źródłowego (w języku *Basic*) na program wynikowy (w języku wewnętrznym), który

zostanie następnie zapisany w pamięci operacyjnej mikrokomputera i wykonany;
— albo do pamięci zostanie wpisany tekst programu źródłowego, a program tłumaczący będzie po kolei odczytywał instrukcje (w języku *Basic*), analizował je i wywoływał odpowiednie podprogramy (w języku wewnętrznym) realizujące wynikające z treści instrukcji działania.

W pierwszym przypadku mówimy o **kompilacji** programu, w drugim o jego **interpretacji**. Zaletą kompilacji jest szybkie wykonanie programu wynikowego — nie traci się już czasu na tłumaczenie instrukcji języka *Basic*. Wadą jest skomplikowany proces wprowadzania zmian i poprawek do programu — każda, nawet najmniejsza zmiana wymaga ponownego tłumaczenia całego programu. W interpretacji przeciwnie: poprawki wprowadza się bardzo łatwo modyfikując tekst programu źródłowego w pamięci, za to wykonanie programu jest powolne. Ze względu na prostszą budowę translatora oraz wygodę użytkowników często uruchamiających nowe programy, mikrokomputery domowe takie jak *Spectrum* lub *Meritum* są standardowo wyposażone w interpretator języka *Basic*, trwale zapisany w wydzielonym obszarze pamięci.

Interpretator języka *Basic* składa się z dwóch oddzielnych części:
— edytora odczytującego tekst programu źródłowego z klawiatury i wpisującego go do pamięci operacyjnej;
— programu interpretującego instrukcje programu źródłowego.

Instrukcje są wpisywane do pamięci w postaci ciągu kodów *ASCII* odpowiadających kolejnym literom i cyfrom tekstu instrukcji. Jedynie słowa kluczowe nie są pamiętane jako wyrazy, lecz zostają zastąpione odpowiednimi kodami liczbowymi. Dodatkowo, każda linia programu źródłowego zostaje uzupełniona adnotacją wskazującą początek następnej linii programu w pamięci operacyjnej (rys. 4.1). Do zadań **edytora** należy więc: odczytywanie tekstu instrukcji pisanych na klawiaturze, rozmieszczanie instrukcji w przeznaczonym na ten cel obszarze pamięci (pole pamięci programu) połączone z ustawianiem ich w kolejności wynikającej z numeracji linii programu, zamiana słów kluczowych na odpowiadające im kody liczbowe. W niektórych implementacjach (np. na *Spectrum*) edytor dodatkowo sprawdza wstępnie poprawność składni — np. obecność nawiasu zamykającego dla każdego nawiasu otwierającego. Zadania **interpretatora** są bardziej skomplikowane i obejmują: pobieranie kolejnych in-



Rys. 4.1. Rozmieszczenie programu w pamięci: *a)* pojedyncza linia programu; *b)* przykład programu

strukcji programu źródłowego z pamięci, analizę treści instrukcji połączoną z wykrywaniem błędów składniowych, rozmieszczanie zmiennych i tablic w przeznaczonym do tego celu obszarze pamięci (pole pamięci zmiennych), wywoływanie podprogramów realizujących działania wynikające z treści instrukcji.

Interpretator języka *Basic* jest trwale zapisany w wydzielonym obszarze pamięci mikrokomputera zbudowanym ze specjalnych elementów pamięciowych o trwale ustalonej zawartości. Dzięki temu interpretator jest zawsze gotowy do pracy, a jego wykonanie rozpoczyna się natychmiast po włączeniu napięcia zasilającego. Cała konwersacja mikrokomputera z użytkownikiem obejmująca wprowadzanie poleceń i wyświetlanie komunikatów jest w istocie prowadzona nie przez mikrokomputer (jako urządzenie), ale przez program interpretatora. **Mikrokomputer, rozumiany jako pewien zestaw sprzętu (*hardware*) nie jest zdolny do żadnych sensownych działań — o wszystkim decydują wprowadzone do jego pamięci programy (*software*).**

Programując mikrokomputer w języku *Basic* wprowadzamy do jego pamięci nowy program, który jest następnie wykonywany (interpretowany). Czynność tę robiliśmy wielokrotnie i wydaje się ona naturalna. Mniej oczywista jest możliwość wprowadzenia do pamięci programu pełniącego rolę translatora języka innego niż *Basic*. Możliwość taka istnieje. **W ciągu ostatnich kilkunastu lat opracowano dla mikrokomputerów translatory wielu języków programowania,** z których szczególną popularnością wyróżniają się języki o nazwach *Pascal* i *Logo*. W porównaniu z językiem *Basic* obydwa wymienione umożliwiają pisanie bardziej uporządkowanych i czytelnych programów. Warto zaznaczyć, że język *Logo* został opracowany specjalne z myślą o zastosowaniach dydaktycznych. Niestety, nawet pobieżne omówienie nowych języków programowania wykracza poza ramy tej książki. Na pocieszenie powiedzmy jednak, że wśród użytkowników mikrokomputerów domowych, takich jak *Spectrum* lub *Meritum*, wciąż najbardziej rozpowszechniony jest język *Basic* i nic nie zapowiada, aby ten stan miał się zmienić.

Podsumowanie

— mikrokomputer wykonuje tylko te działania, które zostały przewidziane w wykonywanym programie;
— mikrokomputer możemy programować w różnych językach programowania, warunkiem jest posiadanie odpowiedniego translatora;
— istnieją dwie zasadnicze techniki translacji: kompilacja i interpretacja.

Pytania

1. Czy instrukcje programu napisanego w języku *Basic*, *Pascal* lub *Logo* stanowią elementarne operacje wykonywane wprost przez procesor mikrokomputera?
2. Jaki sens widzisz w wyposażeniu tego samego mikrokomputera w translatory różnych języków programowania?
3. Dlaczego wykonanie programu tłumaczonego interpretacyjnie trwa kilka razy dłużej niż wykonanie programu skompilowanego?
   (Wskazówka: rozważ program z pętlą, ile razy są tłumaczone instrukcje wewnętrzne pętli?)

4.2
Operacje arytmetyczne na liczbach dwójkowych

Pisanie programów w języku *Basic*, *Pascal* lub *Logo* jest bardzo odległe od programowania w języku wewnętrznym. Istotne różnice występują zarówno na poziomie instrukcji jak i w postaci danych, na których te instrukcje działają. Aby należycie ocenić rolę i znaczenie translatorów, prześledzimy w kolejnych podrozdziałach sposób pisania programów w języku wewnętrznym, bezpośrednio rozumianym przez mikrokomputer.

Elementarną jednostką informacji wewnątrz mikrokomputera jest **bit**, określony jako element, który może przyjmować jedną z dwóch wartości, oznaczanych umownie **0** i **1**. W naturalny sposób bit może być utożsamiony z cyfrą dwójkową. Przy takiej interpretacji ciąg bitów

$$b_n \ldots b_1 b_0$$

worzy liczbę dwójkową o wartości

$$w = b_0 + 2b_1 + \ldots + 2^n b_n = \sum_{i=0}^{n} 2^i b_i$$

Większość operacji wewnętrznych mikrokomputera jest wykonywana nie na pojedynczych bitach, lecz na 8-elementowych ciągach bitów nazywanych **bajtami.** 8-bitowy bajt stanowi podstawową jednostkę informacji przetwarzanej przez komputer. Wartość bajta może być interpretowana w różny sposób, np. jako liczba lub jako kod znaku alfanumerycznego (skrót od alfabetyczno-numeryczny, tzn. litery, cyfry, itp.).

Naturalny zapis dwójkowy jest ograniczony do liczb dodatnich, nazywanych zwyczajowo liczbami bez znaku. Liczby ze znakiem (dodatnie lub ujemne) są zapisywane w kodzie **uzupełnień do dwóch** (w skrócie *U2*). Wartość liczby $b_n \ldots b_1 b_0$ obliczamy tu jako

$$w = b_0 + 2b_1 + \ldots + 2^{n-1} b_{n-1} - 2^n b_n$$

W jednym bajcie możemy w ten sposób zapisać liczby z zakresu −128...+127, np.

00000000 = 0
00000101 = 5
01111111 = 127
10000000 = −128
10000101 = −123 (−128+5)
11111111 = −1 (−128+127)

Najważniejszą cechą zapisu *U2* jest to, że możemy w nim wyrazić zarówno liczby dodatnie jak i ujemne, a ponadto działania dodawania i odejmowania wykonujemy dokładnie tak samo jak odpowiednie działania w naturalnym zapisie dwójkowym. Sprawdźmy dla dodawania

```
  00000101   (= 5)          10000000   (= −128)
+ 00010100   (= 20)       + 01000000   (= 64)
  00011001   (= 25)          11000000   (= −64)

  11100000   (= −32)         11100000   (= −32)
+ 00100001   (= 33)       + 11100000   (= −32)
  00000001   (= 1)           11000000   (= −64)
```

Dzięki tej właściwości zapisu *U2* operacje arytmetyczne wykonywane przez mikroprocesor dają poprawny wynik niezależnie od tego, czy argumenty traktujemy jako liczby w naturalnym zapisie dwójkowym, czy jako liczby ze znakiem w zapisie *U2*.

Ograniczenie liczby bitów używanych do zapisywania liczb sprawia, że wyniki niektórych działań stają się niemożliwe do zapisania. Np. 65 + 65 = 130 — w zapisie *U2* każdy z argumentów może zostać zapisany w jednym bajcie, natomiast wynik już nie

```
  01000001   (= 65)
+ 01000001   (= 65)
  10000010   (= −126)
```

W naturalnym zapisie dwójkowym wynik tego działania (130) jest poprawny. Dla odmiany dodawanie

```
  01000001
+ 11000010
  00000011
```

daje wynik poprawny w zapisie *U2* (65 − 62 = 3), natomiast błędny w naturalnym zapisie dwójkowym (65 + 194 = 259), gdy liczba 259 nie mieści się w 8 bitach.

Sytuację, gdy wynik działania nie mieści się w przyjętej liczbie bitów nazywamy **nadmiarem**. Należy podkreślić, że możliwość powstania nadmiaru nie wynika z żadnej szczególnej cechy zapisu dwójkowego, a jedynie z ograniczenia liczby cyfr używanych do zapisywania wartości. Z taką samą sytuacją spo-

tkamy się w zapisie dziesiętnym, jeżeli ograniczymy liczby np. do czterech cyfr

$$\begin{array}{r} 6000 \\ +6500 \\ \hline 2500 \end{array}$$

Najbardziej znacząca cyfra wyniku ginie i rezultat jest błędny.

W czasie wykonywania działań arytmetycznych mikroprocesor sprawdza przebieg operacji i wykrywa ewentualne powstanie nadmiaru. Reguły pozwalające wykryć nadmiar są w naturalnym zapisie dwójkowym inne niż w zapisie *U2*. Z tego powodu mikroprocesor wytwarza zawsze dwa sygnały ostrzegawcze:

— nadmiar *U2*,

— nadmiar w naturalnym zapisie dwójkowym.

Zależnie od tego jak interpretujemy dane, korzystamy z pierwszego lub drugiego sygnału ostrzegawczego.

Rozważmy nieco dokładniej nadmiar dodawania w naturalnym zapisie dwójkowym, w którym ma on wyjątkowo przejrzystą interpretację. Sygnałem nadmiaru jest po prostu bit przeniesienia wychodzącego z najbardziej znaczącej pozycji, np.

$$\begin{array}{rr} 00000101 & 01000001 \\ +\ \underline{00010100} & +\ \underline{11000010} \\ 0 \leftarrow 00011001 & 1 \leftarrow 00000011 \\ \text{wynik poprawny} & \text{nadmiar} \end{array}$$

Przeniesienie, które będziemy dalej oznaczać skrótem *CY* (od ang. *Carry*), możemy wykorzystać do realizacji dodawania liczb zapisanych w kilku bajtach

$$\begin{array}{rrl} & 1 & \\ 00001000 & 00100000 & (= 2080) \\ +\ \underline{00000000} & \underline{11100000} & (= 224) \\ 0 \leftarrow 00001001 & 00000000 & (= 2304) \end{array}$$

Dodawanie liczb zapisanych w dwóch bajtach wykonujemy w dwóch krokach. W pierwszym kroku dodajemy mniej znaczące bajty liczb, w drugim bardziej znaczące bajty i bit przeniesienia z poprzedniej operacji.

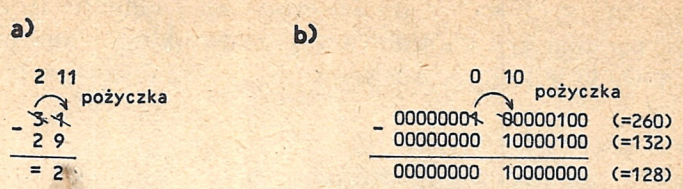
Dodawanie liczb zapisanych w kilku bajtach w kodzie *U2* wykonujemy według dokładnie tej samej reguły, np.

$$\begin{array}{rrl} & 1 & \\ 11111111 & 10000000 & (= -128) \\ +\ \underline{00000000} & \underline{11100000} & (= 224) \\ 00000000 & 01100000 & (= 96) \end{array}$$

Nadmiar całej operacji sprawdzamy dopiero po dodaniu najbardziej znaczących bajtów.

Nadmiar, czyli niemożliwość wyrażenia wyniku w przyjętej formie zapisu może wystąpić również podczas odejmowania. W naturalnym zapisie dwójkowym ma to miejsce wówczas, gdy próbujemy odjąć liczbę większą od mniej-

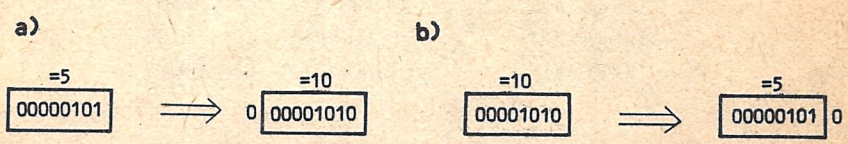
szej. Nadmiar, który będziemy tak jak poprzednio oznaczać skrótem *CY*, ma tu znaczenie pożyczki w takim sensie, jak to rozumiemy podczas odejmowania liczb dziesiętnych (rys. 4.2). **Przy odejmowaniu liczb zapisanych w dwóch bajtach działanie wykonujemy w dwóch krokach; w pierwszym odejmujemy mniej znaczące bajty, w drugim bardziej znaczące bajty i bit pożyczki (przeniesienia *CY*).**



Rys. 4.2. Odejmowanie: *a)* liczby dziesiętne; *b)* liczby dwójkowe

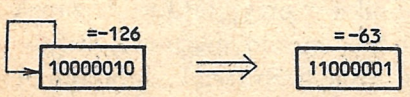
Odejmowanie liczb zapisanych w kilku bajtach w kodzie *U2* wykonujemy dokładnie według tej samej reguły. Nadmiar całej operacji sprawdzamy dopiero po odjęciu najbardziej znaczących bajtów.

Mnożenie i dzielenie są operacjami znacznie bardziej skomplikowanymi niż dodawanie i odejmowanie. Mikroprocesory mikrokomputerów *Spectrum* i *Meritum* nie są w stanie tych działań wykonać — są one realizowane przez odpowiednie podprogramy. Wyjątkiem jest **mnożenie i dzielenie przez 2, które wykonuje się przez dopisanie lub usunięcie ostatniego zera,** tak samo jak mnożenie i dzielenie liczb dziesiętnych przez 10. Jeżeli prowadzimy operacje na liczbach o stałej długości (np. 8 bitów), to dopisanie lub usunięcie zera wygodnie jest traktować jako operację przesunięcia liczby (rys. 4.3). Przesunięcie w



Rys. 4.3. Mnożenie i dzielenie przez 2: *a)* dopisanie zera, czyli przesunięcie w lewo; *b)* usunięcie zera, czyli przesunięcie w prawo

lewo realizuje mnożenie liczby przez 2, przesunięcie w prawo dzielenie przez 2. Nieco bardziej złożone jest dzielenie przez 2 liczb ujemnych zapisanych w kodzie *U2*. „Zwykłe" przesunięcie w prawo spowoduje zmianę znaku. Aby prawidłowo podzielić liczbę ujemną przez 2, musimy na najbardziej znaczącą pozycję wprowadzić nie **0** lecz **1** (rys. 4.4).



Rys. 4.4. Dzielenie liczby ujemnej przez 2

Podsumowanie

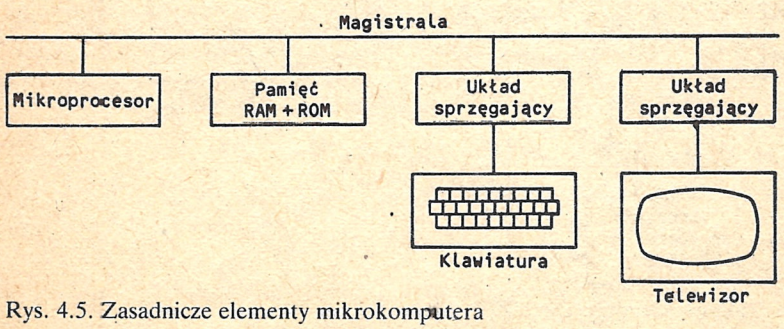
— mikroprocesor wykonuje operacje arytmetyczne na liczbach bez znaku w naturalnym zapisie dwójkowym i na liczbach ze znakiem w zapisie *U2*;
— po wykonaniu działania może pojawić się nadmiar — sytuacja, gdy wynik nie mieści się w przyjętej liczbie bitów;
— działania na liczbach zapisanych w kilku bajtach wykonujemy w kilku krokach, korzystając z bitu przeniesienia;
— mnożenie przez 2 realizujemy przez przesunięcie liczby w lewo, dzielenie przez 2 realizujemy przez przesunięcie liczby w prawo, z powieleniem bitu znaku.

Pytania i zadania

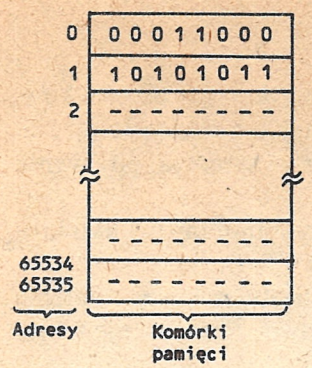
1. Napisz w języku *Basic* podprogram wprowadzania liczb w zapisie *U2* (por. p. 2.4).
2. Liczby x i y są pamiętane w trzech bajtach każda: $x = abc$, $y = def$. Narysuj sieć działań dodawania $x = x + y$, korzystając z operacji dodawania pojedynczych bajtów i przeniesienia;
3. Czy w czasie przesuwania w lewo może pojawić się nadmiar?

4.3
Budowa mikrokomputera

Podstawowymi zespołami składającymi się na mikrokomputer są: mikroprocesor, pamięć operacyjna (czyli wewnętrzna) i układy sprzęgające z urządzeniami zewnętrznymi. Wszystkie zespoły są połączone przewodami tworzącymi magistralę mikrokomputera (rys. 4.5).

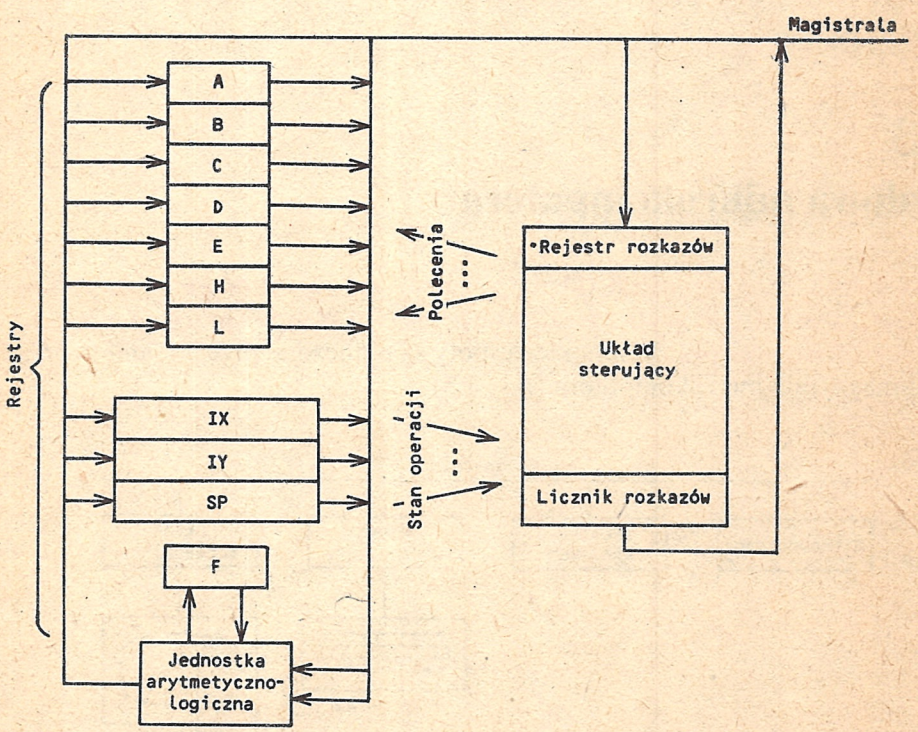


Rys. 4.5. Zasadnicze elementy mikrokomputera



Rys. 4.6. Pamięć operacyjna

Informacja przetwarzana przez mikrokomputer jest gromadzona w bloku pamięci. Z punktu widzenia użytkownika **pamięć operacyjną** możemy traktować jako zbiór komórek pamięci, z których każda może zmieścić pojedynczy bajt. Wszystkie komórki wchodzące w skład pamięci są ponumerowane kolejnymi liczbami całkowitymi, poczynając od liczby 0. Numer komórki nazywa się jej **adresem** w bloku pamięci (rys. 4.6). Liczba komórek pamięci (tzw. **pojemność pamięci**) jest ważnym parametrem określającym możliwości komputera. W popularnych mikrokomputerach domowych pojemność pamięci wynosi od 32k bajtów do 128k bajtów, przy czym 1k bajt (jeden kilobajt) jest równy 1024 komórkom pamięci ($= 2^{10}$). Część tej pojemności jest trwale wydzielona i zawiera interpretator języka *Basic*. Na przykład mikrokomputer *Meritum-1* ma pamięć o pojemności 16k bajtów plus 16k bajtów zajętych przez interpretator języka *Basic*, a mikrokomputer *Spectrum* 48k i *Spectrum Plus* 48k bajtów plus 16k bajtów zajętych przez interpretator języka *Basic*. Zarówno zapisywanie danych w pamięci jak i odczytywanie danych z pamięci jest dokonywane zawsze całymi bajtami.



Rys. 4.7. Struktura mikroprocesora *Z80*

Centralnym elementem mikrokomputera realizującym przetwarzanie informacji jest **mikroprocesor**. Z technologicznego punktu widzenia mikroprocesor jest niepodzielnym elementem półprzewodnikowym wykonanym w technologii wielkiej skali integracji (*LSI*). Funkcjonalnie, możemy wyróżnić wewnątrz mikroprocesora trzy podukłady (rys. 4.7): jednostkę arytmetyczno-logiczną, zespół rejestrów i układ sterujący. **Zespół rejestrów** zawiera kilka lub kilkanaście komórek pamięci tworzących lokalną, podręczną pamięć mikroprocesora. Komórki pamięci znajdujące się wewnątrz mikroprocesora nazywa się rejestrami. Ilość informacji, jaka może być pamiętana w rejestrach, zależy od typu mikroprocesora.

W mikrokomputerach *Spectrum* i *Meritum* jest wykorzystany mikroprocesor *Z80* zawierający 8 rejestrów 8-bitowych oznaczanych *A, B, C, D, E, H, L* i *F* oraz dodatkowo 3 rejestry 16-bitowe oznaczone *IX*, *IY* i *SP*. Rejestry 8-bitowe mogą być używane również w parach: *BC*, *DE*, *HL*, tworząc wówczas 3 rejestry 16-bitowe. **Jednostka arytmetyczno-logiczna** stanowi układ wykonujący elementarne operacje na danych, takie jak: dodawanie, odejmowanie, dodawanie i mnożenie logiczne, przesuwanie zawartości rejestrów. Większość operacji jest wykonywana na argumentach 8-bitowych. W większości działań jeden z argumentów jest pobierany z rejestru *A* (nazywanego **akumulatorem**), a drugi — jeżeli jest potrzebny — z jednego z pozostałych rejestrów lub z komórki pamięci. Wynik operacji jest z reguły wpisywany do rejestru *A*. Szczególną rolę pełni też rejestr *F*, nazywany **rejestrem znaczników**. Poszczególne bity tego rejestru są wykorzystane do sygnalizacji pewnych szczególnych cech wyniku operacji arytmetycznych, takich jak:

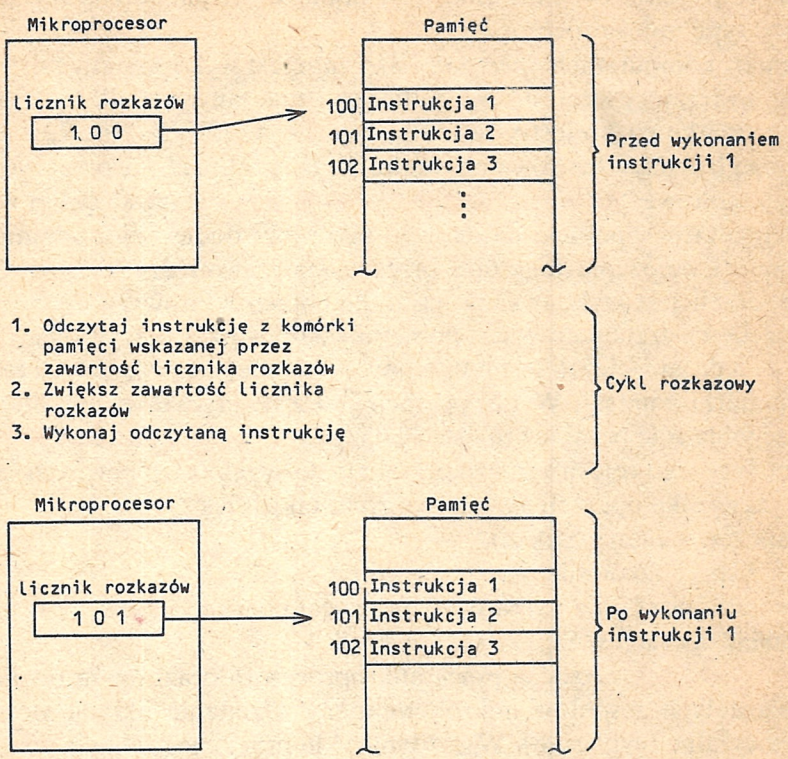
— wynik zerowy (znacznik *Z*),
— znak wyniku (znacznik *S*),
— przeniesienie (*CY*) z najbardziej znaczącej pozycji,
— nadmiar operacji w zapisie *U2* (znacznik *V*).

Układ sterujący zarządza pracą mikroprocesora oraz współpracujących z nim pozostałych zespołów mikrokomputera. Działanie układu sterującego jest powtarzalne (cykliczne). W każdym cyklu pracy, nazywanym **cyklem rozkazowym**, jest wykonywana pojedyncza instrukcja języka wewnętrznego.

Zbiór wszystkich instrukcji nosi nazwę **listy rozkazów** mikroprocesora. Różne typy mikroprocesorów mają różne listy rozkazów. Czas wykonania typowych instrukcji wynosi 1...5 μs, co odpowiada szybkości pracy ok. 500 000 instrukcji na sekundę.

Z cykliczności pracy układu sterującego wynika zdolność do wykonywania nie tylko pojedynczych instrukcji, ale i całych ciągów instrukcji tworzących program. Kolejne instrukcje programu są rozmieszczone w kolejnych komórkach pamięci, przy czym pewne instrukcje mieszczą się w całości w jednej komórce, a inne mogą zajmować dwie, trzy, a nawet cztery komórki pamięci. Adres instrukcji, która powinna być wykonana w danym cyklu rozkazowym jest umieszczony w specjalnym rejestrze **licznika rozkazów** wewnątrz układu sterującego. Po odczytaniu całego rozkazu z pamięci zawartość licznika roz-

kazów zostaje automatycznie zwiększona o liczbę równą liczbie bajtów zajmowanych przez odczytaną instrukcję. Dzięki posiadaniu licznika rozkazów mikroprocesor zawsze wie, z której komórki pamięci należy pobrać instrukcję do wykonania. Dzięki automatycznemu zwiększaniu zawartości licznika rozkazów, w kolejnych cyklach rozkazowych są wykonywane kolejne instrukcje programu (rys. 4.8). Specjalne instrukcje umożliwiają zapis nowej wartości do licznika rozkazów, co powoduje zmianę kolejności wykonania instrukcji (skok w programie).



Rys. 4.8. Adresowanie instrukcji programu w pamięci

Układy sprzęgające urządzeń zewnętrznych pracują pod nadzorem mikroprocesora. Ich zadanie polega na umożliwieniu mikroprocesorowi przesyłania danych do urządzenia lub odczytywania danych z urządzenia. Podobnie jak przy współpracy mikroprocesora z pamięcią dane są przesyłane zawsze całymi bajtami. To znaczy, w liście rozkazów mikroprocesora znajdują się specjalne instrukcje typu: „czytaj bajt z urządzenia" oraz „pisz bajt do urządzenia" powodujące przesłanie bajta danych między jednym z rejestrów mikroprocesora (wskazanym w treści instrukcji) a jednym z urządzeń (również wskazanym w treści instrukcji). W celu rozróżnienia urządzeń, każde z nich ma własny, niepowtarzalny numer. Niektóre układy sprzęgające o specjalnej konstrukcji (tzw. układy z **bezpośrednim dostępem do pamięci**) umożliwiają przesyła-

nie wielu bajtów danych między urządzeniem a wyznaczonym obszarem w pamięci operacyjnej. Zadaniem układu sprzęgającego jest tu samodzielne sterowanie przebiegiem przesyłania kolejnych bajtów bloku danych.

Z bezpośrednim dostępem do pamięci pracuje w mikrokomputerach **układ sprzęgający współpracujący z monitorem lub telewizorem**. W wydzielonym polu pamięci operacyjnej pamiętany jest obraz, konstruowany przez mikroprocesor w czasie wykonywania programu. Układ sprzęgający okresowo odczytuje zawartość pola pamięci obrazu i wyświetla go na ekranie. Częstotliwość wyświetlania obrazu wynika ze standardów przyjętych w odbiornikach telewizyjnych i wynosi 50 razy na sekundę. Szczegółowy sposób wyświetlania obrazu jest w różnych komputerach różny. W *Spectrum* w polu pamięci obrazu jest przechowywana dokładna mapa ekranu, w której każdemu elementarnemu punktowi ekranu jest przypisany jeden bit o wartości: **0** — gdy punkt jest w kolorze papieru, **1** — gdy punkt jest w kolorze atramentu. Pole pamięci obrazu zajmuje w pamięci mikrokomputera *Spectrum* obszar 6k bajtów o adresach od 16384 do 22527. W mikrokomputerze *Meritum* organizacja wyświetlania jest zupełnie inna. W polu pamięci obrazu są pamiętane tylko kody *ASCII* znaków, które mają być wyświetlone. Układ sprzęgający monitora okresowo (50 razy na sekundę) odczytuje zawartość pola pamięci obrazu, na podstawie kodów generuje kropkowe wzorce znaków i wyświetla je na ekranie. Dzięki takiemu rozwiązaniu pole pamięci obrazu jest znacznie mniejsze (8 bitów na znak — zamiast 64) i zajmuje obszar tylko 1k bajta, o adresach od 15360 do 16383. Niestety, zysk ten jest opłacony rezygnacją z grafiki wysokorozdzielczej — mikroprocesor może wyświetlać tylko znaki z repertuaru przewidzianego w konstrukcji układu sprzęgającego.

Podsumowanie

— podstawowymi elementami mikrokomputera są: mikroprocesor, pamięć i układy sprzęgające urządzeń zewnętrznych;
— mikroprocesor pobiera z pamięci i wykonuje kolejne instrukcje programu, zapisane w języku wewnętrznym;
— wewnątrz mikroprocesora możemy funkcjonalnie wyróżnić: zespół rejestrów, jednostkę arytmetyczno-logiczną i układ sterujący.

Pytania i zadania

1. Wyjaśnij rolę poszczególnych bloków mikrokomputera.
2. Czy można naprawić zepsuty rejestr wewnętrzny mikroprocesora?
3. Wyjaśnij rolę rejestru akumulatora i rejestru znaczników.
4. W mikroprocesorze *Z80* adresy komórek pamięci są zawsze liczbami 16-bitowymi. Jaka jest maksymalna pojemność pamięci operacyjnej, z którą może współpracować ten mikroprocesor?

4.4
Lista rozkazów mikroprocesora Z80

Jeśli wierzyć katalogom, lista rozkazów mikroprocesora *Z80* zawiera 158 instrukcji, które możemy podzielić na sześć grup:
— instrukcje przesłań zawartości rejestrów i komórek pamięci;
— instrukcje operacji arytmetycznych i logicznych;
— instrukcje operacji na pojedynczych bitach;
— instrukcje skoków, skoków warunkowych i wywołań podprogramów;
— instrukcje komunikacji z urządzeniami zewnętrznymi;
— różne instrukcje pomocnicze.
Każdą grupę omówimy oddzielnie, posługując się wybranymi przykładami typowych instrukcji.

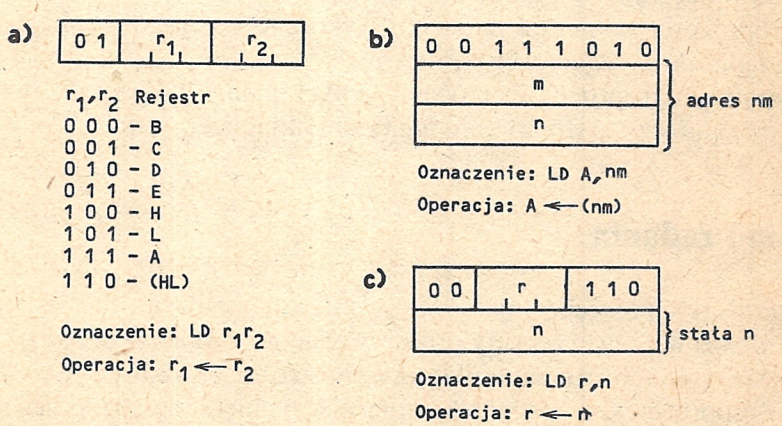
1. **Instrukcje przesłań** pozwalają na przesłanie zawartości jednego rejestru do dowolnego innego rejestru, zawartości rejestru do komórki pamięci lub zawartości komórki pamięci do rejestru. Najprostszą postać mają rozkazy przesłań między rejestrami, realizujące operację

$$r_1 \leftarrow r_2$$

w wyniku której nową zawartością rejestru r_1 staje się zwartość rejestru r_2. Zawartość rejestru r_2 nie ulega zmianie (analogicznie jak w instrukcji podstawienia w języku *Basic*). Instrukcje przesłania między rejestrami mieszczą się w pojedynczym bajcie pamięci (rys. 4.9*a*). Bardziej złożony format (czyli sposób zapisu w pamięci) mają instrukcje przesłań między rejestrami a komórkami pamięci, realizujące operacje

$$r \leftarrow (x) \quad \text{lub} \quad (x) \leftarrow r$$

gdzie (x) oznacza zawartość komórki pamięci o adresie x. Adres ten może być określony w różny sposób.



Rys. 4.9. Instrukcje przesłań: *a)* rejestr-rejestr; *b)* pamięć-rejestr akumulatora; *c)* ładowanie stałej do rejestru

a) Adresem może być zawartość pary rejestrów *HL* (rys. 4.9*a*). Na przykład w rozkazie *LD B,(HL)* do rejestru *B* zostanie przesłana zawartość komórki pamięci o adresie równym 16-bitowej liczbie zapisanej w rejestrach *HL*; zawartość pamięci nie ulegnie zmianie.

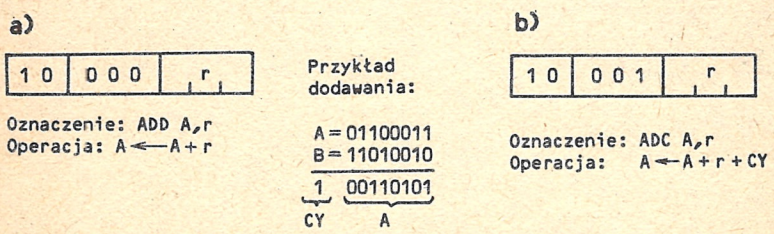
b) W przypadku przesłań z lub do rejestru *A* (akumulatora), adres komórki pamięci może być podany wprost w treści instrukcji (rys. 4.9*b*). Formaty rozkazów *LD A,(adres)* i *LD (adres),A* różnią się tylko wartością jednego bitu w pierwszym bajcie instrukcji.

c) Do rejestru może być przesłana stała, zawarta w treści instrukcji (rys. 4.9*c*).

Oprócz przesłań pojedynczych bajtów w liście rozkazów mikroprocesora *Z80* jest zawarte kilkanaście instrukcji przesłania zawartości rejestrów 16-bitowych (lub par rejestrów) do lub z dwóch sąsiednich komórek pamięci

2. **Instrukcje arytmetyczne i logiczne** umożliwiają wykonanie działań dodawania i odejmowania liczb 8-bitowych i 16-bitowych oraz wykonanie operacji logicznych na argumentach 8-bitowych. Operacje 8-bitowe są wykonywane zawsze na zawartości rejestru *A* (pierwszy argument) i zawartości dowolnego innego rejestru lub komórki pamięci adresowanej przez parę *HL* (drugi argument — rys. 4.10). Drugim argumentem operacji może być też stała podana w treści instrukcji. Wynik operacji jest wpisywany zawsze do rejestru *A*. Szczególne cechy wyniku są zapamiętywane w znacznikach rejestru *F*. Na przykład instrukcja *ADD A,B* wykonana na zawartościach rejestrów pokazanych na rys. 4.10*a* powoduje obliczenie sumy liczb dwójkowych 01100011 + 11010010, wpisanie wyniku 00110101 do rejestru *A* i ustawienie znaczników rejestru *F*

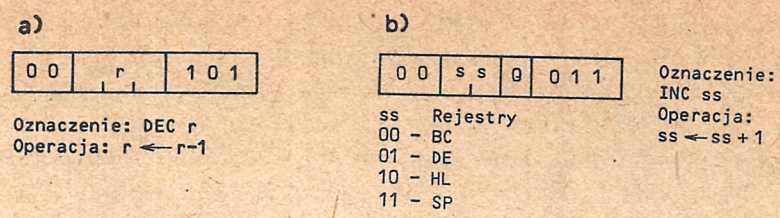
CY = 1 — przeniesienie z najbardziej znaczącej pozycji,
S = 0 — wynik jest dodatni,
Z = 0 — wynik nie jest zerowy,
V = 0 — brak nadmiaru *U2*.



Rys. 4.10. Instrukcja dodawania: *a)* liczb 8-bitowych; *b)* liczb 8-bitowych i przeniesienia *CY*

Zawartość rejestru *B* pozostaje nie zmieniona. Oprócz instrukcji dodawania i odejmowania dowolnych argumentów, w liście rozkazów znajdują się specjalne instrukcje dodawania i odejmowania jedynki (rys. 4.11).

Operacje logiczne, takie jak dodawanie logiczne, mnożenie logiczne i

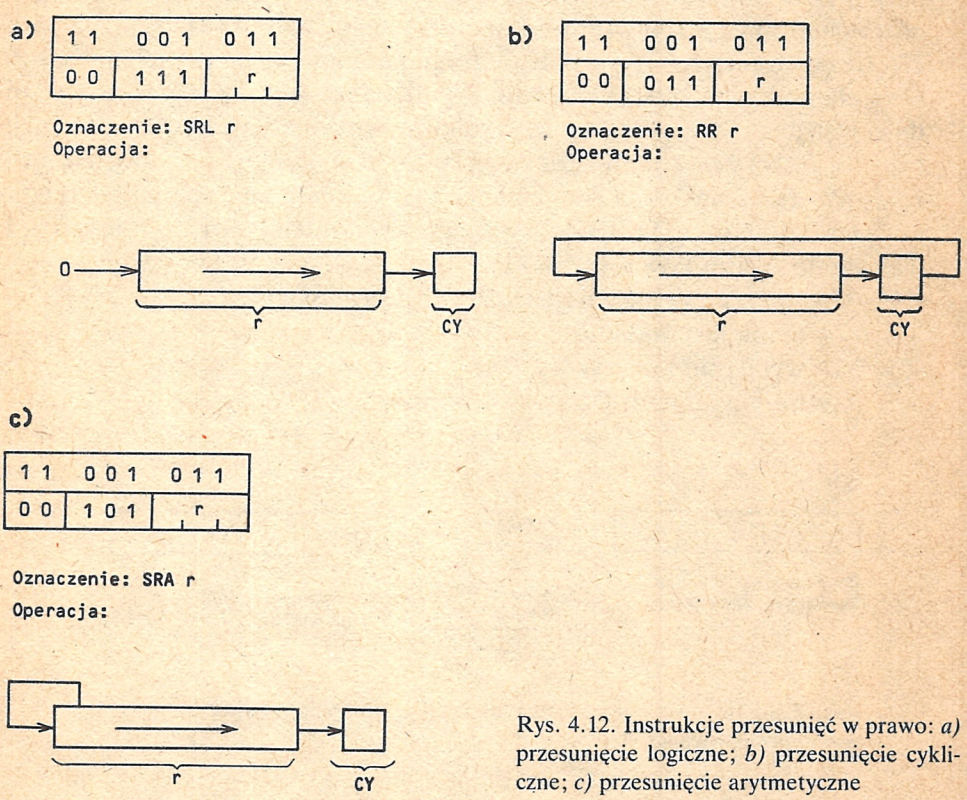


Rys. 4.11. Instrukcje zmiany zawartości rejestru o 1: *a)* zmniejszenie zawartości rejestru 8-bitowego; *b)* zwiększenie zawartości pary rejestrów

suma modulo dwa są wykonywane na parach odpowiadających sobie bitów argumentów, np.

$$\begin{array}{rr} 01100110 & 01100110 \\ \wedge\ 00001111 & \vee\ 00001111 \\ \hline 00000110 & 01101111 \end{array}$$

Sposób adresowania argumentów tych operacji jest taki sam jak w instrukcjach działań arytmetycznych.



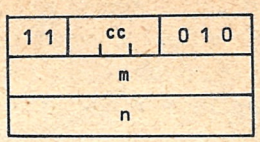
Rys. 4.12. Instrukcje przesunięć w prawo: *a)* przesunięcie logiczne; *b)* przesunięcie cykliczne; *c)* przesunięcie arytmetyczne

Ważną grupę instrukcji stanowią również operacje przesunięć (rys. 4.12). Są to operacje jednoargumentowe zmieniające zawartość rejestru lub komórki pamięci zaadresowanej przez zawartość pary rejestrów *HL*.

Oprócz pokazanych na rysunku przesunięć w prawo, w liście rozkazów znajdują się również analogiczne operacje przesunięcia w lewo.

3. **Operacje na bitach** pozwalają na indywidualne ustawianie (w stan 1) lub zerowanie, a także sprawdzanie wartości poszczególnych bitów w rejestrach lub w komórkach pamięci. Zauważmy, że zmianę wartości pojedynczych bitów możemy też łatwo zaprogramować za pomocą operacji dodawania i mnożenia logicznego.

4. **Instrukcje skoków** pozwalają na zmianę kolejności wykonania instrukcji programu (podobnie jak instrukcja *GOTO* w języku *Basic*). Oprócz instrukcji bezwarunkowego wykonania skoku w liście rozkazów są również instrukcje skoków warunkowych, stanowiące odpowiednik instrukcji *IF...THEN GOTO...* w języku *Basic*. Argumentem instrukcji skoku jest adres instrukcji, która ma być wykonana jako następna. W instrukcjach skoku warunkowego skok jest wykonywany lub nie, zależnie od wartości poszczególnych bitów rejestru *F*. Odrębne instrukcje (rys. 4.13) powodują skok, gdy ustawiony (albo wyzerowany) jest wskaźnik przeniesienia *CY*, wskaźnik zera *Z*, wskaźnik wyniku ujemnego *S* lub wskaźnik nadmiaru *V*. Szczególnym rodzajem skoku jest skok do podprogramu o działaniu podobnym do instrukcji *GOSUB* w języku *Basic*. Powrót z podprogramu do programu głównego powoduje instrukcja *RET*.



cc	oznaczenie	warunek
000	NZ	Z=0
001	Z	Z=1
010	NC	CY=0
011	C	CY=1
100	PO	V=0
101	PE	V=1
110	P	S=0
111	M	S=1

Z,CY,V,S - bity rejestru F

Oznaczenie instrukcji: JP cc,nm
Operacja:
JEŻELI cc TO SKOCZ DO nm

Rys. 4.13. Instrukcje skoków warunkowych

5. **Instrukcje komunikacji** z urządzeniami zewnętrznymi są podobne do instrukcji przesłań. Wykonanie instrukcji powoduje na ogół przesłanie pojedynczego bajta danych między jednym z rejestrów mikroprocesora a wybranym urządzeniem. Adres (numer) urządzenia jest podawany w treści instrukcji.

6. **Instrukcje pomocnicze** stanowią nieliczną grupę instrukcji wykonujących pewne funkcje organizacyjne takie, jak np. zmiana wartości wskaźnika *CY* lub zmiana znaku liczby zapisanej w akumulatorze w kodzie *U2*.

Zanim zaczniemy pisać programy w języku wewnętrznym i zmieniać zawartości komórek pamięci, musimy ustalić, w jakim obszarze pamięci będziemy działać tak, aby nie zakłócić działania interpretatora języka *Basic*. Na szczęście w konstrukcji interpretatora przewidziano taką możliwość. Interpretator wykorzystuje zawsze pamięć od początku (od adresu 0) aż do końca, przy czym położenie końca pamięci możemy określić dowolnie. W mikrokomputerze *Meritum* musimy to zrobić od razu na początku, po pierwszym komunikacie

OBSZAR PAO?-
po którym możemy nacisnąć klawisz *ENTER*, udostępniając dla interpretatora całą pamięć, albo napisać adres ostatniej komórki obszaru pamięci przeznaczonego dla interpretatora. W mikrokomputerze *Spectrum* koniec obszaru dostępnego dla interpretatora określamy za pomocą polecenia
CLEAR m
gdzie *m* jest adresem końca obszaru pamięci dostępnego dla interpretatora. W dalszych przykładach przyjmiemy, że adres ostatniej dostępnej dla interpretatora komórki pamięci ma wartość 31999.

Poznane w rozdz. 2 instrukcje języka *Basic* nie dają żadnych możliwości programowania w języku wewnętrznym. Nasz mikrokomputer wykonuje jednak kilka instrukcji (dołączonych do języka *Basic*), które pozwolą na wprowadzenie i uruchomienie krótkich programików. Instrukcje te umożliwiają zapisywanie i odczytywanie konkretnych komórek pamięci oraz wywoływanie podprogramów napisanych w języku wewnętrznym i wprowadzonych do pamięci.

Instrukcja *POKE* o postaci
POKE x, y
powoduje wpisanie liczby *y* do komórki pamięci o adresie *x*. Oczywiście, wartość liczby powinna zawierać się w zakresie $y = 0...255$. Użytkownicy mikrokomputera *Spectrum* mogą podawać wartość liczby *y* również w zapisie dwójkowym, używając funkcji *BIN*, tak jak to zrobiliśmy w p. 2.7. **Funkcja *PEEK (x)*** ma działanie odwrotne. Jej wartością jest zawartość komórki pamięci o adresie *x*. Na przykład polecenie
PRINT PEEK (16384)
spowoduje wyświetlenie na ekranie zawartości komórki o adresie 16384.

Wywołanie podprogramu napisanego w języku wewnętrznym umożliwia **funkcja *USR***, która działa w obydwu implementacjach języka *Basic* nieco inaczej. W mikrokomputerze *Meritum* parametrem funkcji *USR* jest nazwa zmiennej, której wartość jest przekazywana do wywoływanego podprogramu. Wartością funkcji może stać się zawartość pary rejestrów *HL* w chwili zakończenia podprogramu. Adres podprogramu, tzn. adres pierwszej instrukcji podprogramu powinien zostać wcześniej umieszczony w komórkach pamięci o adresach 16526, 16527. To znaczy wywołanie o postaci
Z = USR (X)
spowoduje rozpoczęcie wykonania podprogramu, którego adres jest zapisany w komórkach 16526, 16527. Wywołanie teraz (wewnątrz podprogramu w języku wewnętrznym) podprogramu specjalnego
CALL 2687
przepisze wartość zmiennej *X* do pary rejestrów *HL*. Po zakończeniu wykonania podprogramu instrukcją maszynową *RET* wartością funkcji *USR* jest wartość parametru *X*. Po zakończeniu wykonania podprogramu skokiem do specjalnej sekwencji końcowej
JP 2714

wartością funkcji *USR* stanie się ostatnia zawartość pary rejestrów *HL* (obliczona przez podprogram).

W mikrokomputerze *Spectrum* parametrem funkcji *USR* jest adres początku podprogramu. Wartością funkcji jest zawartość pary rejestrów *BC* w chwili zakończenia podprogramu. Drugą postać funkcji *USR* w mikrokomputerze *Spectrum* poznaliśmy w p. 2.7. Argumentem funkcji był tam jednoznakowy łańcuch zawierający nazwę definiowanego symbolu graficznego. Wartością tak wywołanej funkcji był adres pierwszego z ośmiu bajtów mieszczących kropkowy wzorzec symbolu.

Podsumowanie

1. Instrukcje języka wewnętrznego działają na poziomie technicznych elementów mikrokomputera: rejestrów, znaczników i komórek pamięci.
2. Komunikacja między poziomem języka *Basic* a poziomem języka wewnętrznego jest możliwa przy użyciu instrukcji *POKE* oraz funkcji *PEEK* i *USR*.

Pytania i zadania

1. Zmienne języka *Basic* są przechowywane w komórkach pamięci o znanych adresach. Jakie instrukcje języka wewnętrznego są potrzebne do realizacji instrukcji podstawienia *LET A = B* ?
2. Napisz w języku wewnętrznym podprogram dzielenia liczby bez znaku zapisanej w komórkach pamięci o adresach 32000 (mniej znaczący bajt) i 32001 (bardziej znaczący bajt) przez 2.
3. Napisz w języku *Basic* program, który wprowadzi do pamięci i wywoła podprogram napisany w pyt. 2.
4. Napisz w języku wewnętrznym podprogram dodawania liczb dwubajtowych zapisanych w komórkach pamięci o adresach: 32000, 32001 (pierwszy składnik) i 32002, 32003 (drugi składnik).
5. Napisz w języku *Basic* program sumatora korzystający z podprogramu napisanego w pyt. 4.

4.5
Interpretator języka Basic

Instrukcje języka *Basic* są wykonywane przez odpowiednie podprogramy napisane w języku wewnętrznym. Aby zrozumieć sposób działania translatora, przedstawimy w zarysie realizację działań i funkcji arytmetycznych przy użyciu elementarnych operacji dostępnych w języku wewnętrznym.

Pierwszym problemem jest sposób przechowywania zmiennych w pamięci. Jak pamiętamy z rozdz. 2 każda zmienna jest określona przez dwa atrybuty: nazwę i wartość. Obydwa są przechowywane w polu pamięci zmiennych interpretatora. **Każda zmienna zajmuje kilka kolejnych bajtów zawierających: nazwę zmiennej (ciąg liter nazwy w kodzie *ASCII*) oraz wartość. Wartość zmiennej jest pamiętana w postaci wykładniczej**

$$\pm m \cdot 2^{\pm e}$$

To znaczy, w kolejnych bajtach są pamiętane: wartość wykładnika e zapisana w kodzie *U2* oraz wartość mantysy m również zapisana w kodzie *U2*. Wykładnik zajmuje na ogół jeden bajt (zakres $-128 \ldots +127$), mantysa — zależnie od implementacji — trzy (*Meritum*) lub cztery (*Spectrum*) bajty. Każda zmienna prosta zajmuje więc w pamięci cztery lub pięć bajtów (wartość) plus bajty przeznaczone na zapisanie nazwy.

Sposób wykonywania działań arytmetycznych na wartościach zmiennych wynika z reguł arytmetyki.

Oznaczmy

$$x = m \cdot 2^e$$
$$y = n \cdot 2^a$$

i przypuśćmy, że $e > a$ i $b = e - a$. Wtedy

$x \cdot y = (m \cdot n)2^{(e+a)}$

$x : y = (m : n)2^{(e-a)}$

$x + y = m2^e + n2^a = m2^e + n2^{a-e+e} = m2^e + n2^{(a-e)}2^e =$
$= m2^e + n2^{-b}2^e = (m + n2^{-b})2^e$

$x - y = (m - n2^{-b})2^e$

Działania na liczbach zapisanych w postaci wykładniczej rozbijają się więc na proste działania arytmetyczne wykonywane oddzielnie na mantysach i wykładnikach:

— dodawanie i odejmowanie;
— dzielenie przez 2 (bo $n2^{-b} = n : 2 : 2 \ldots : 2$ b razy);
— mnożenie i dzielenie.

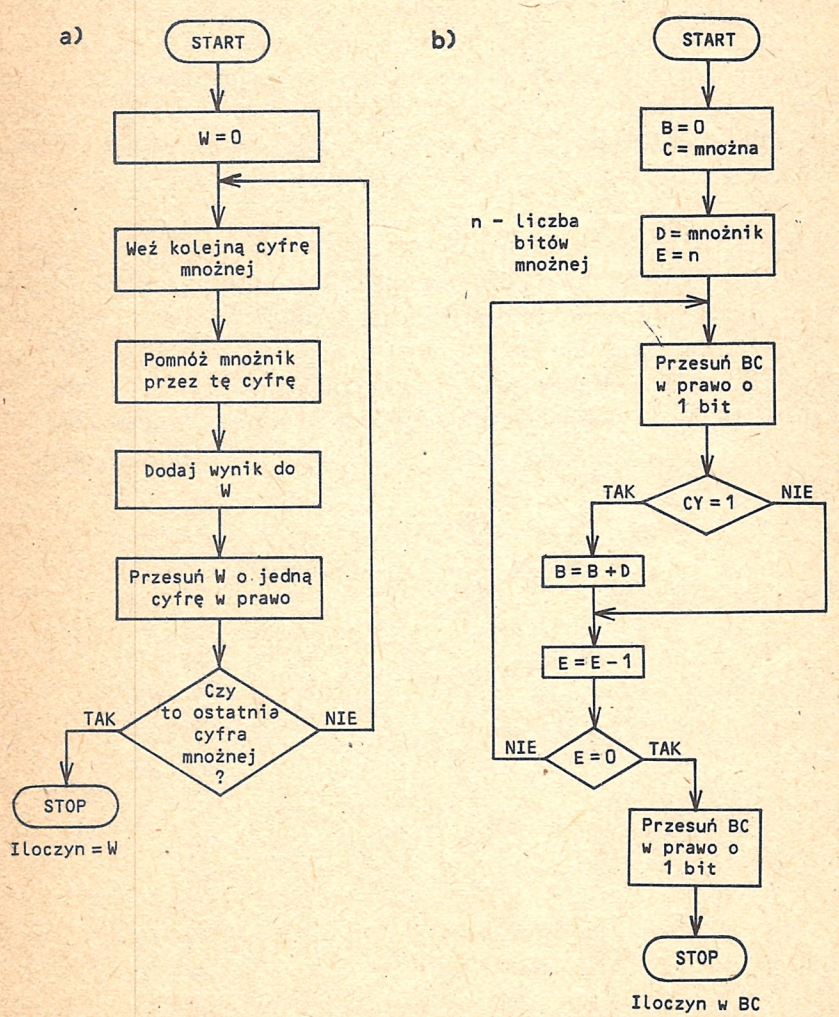
Trzy pierwsze działania potrafimy zrealizować za pomocą instrukcji języka wewnętrznego. Z dwóch pozostałych przedstawimy sposób wykonania operacji mnożenia.

Najprostszy algorytm wynika wprost z definicji mnożenia liczb naturalnych: pomnożyć liczbę a przez liczbę b, znaczy dodać liczbę a do siebie b

razy. Algorytm ten jest jednak nieprzydatny ze względu na bardzo długi czas jego wykonania. Znacznie szybciej możemy obliczyć iloczyn posługując się a gorytmem stosowanym przy mnożeniu wielocyfrowym liczb dziesiętnych na papierze

```
   345      mnożnik
 × 957      mnożna
  2415  =  345 × 7 przesunięte 3 razy w prawo
  1725  =  345 × 5 przesunięte 2 razy w prawo
+ 3105  =  345 × 9 przesunięte 1 raz w prawo
330165  =  345 × 957
```

W algorytmie tym mnożymy mnożnik przez kolejne cyfry mnożnej i sumujemy wyniki tych mnożeń poprzesuwane względem siebie o jedną pozycję (cyfrę). Jeżeli umówimy się, że w podstawowym położeniu wynik mnożenia mno-



Rys. 4.14. Algorytm mnożenia: *a)* liczb dziesiętnych; *b)* liczb dwójkowych

żnika przez cyfrę mnożnej znajduje się bezpośrednio przed najbardziej znaczącą cyfrą mnożnej, to możemy powiedzieć, że wyniki kolejnych mnożeń są poprzesuwane w prawo. Dodanie i przesunięcie każdego kolejnego wyniku możemy zrealizować bezpośrednio po jego obliczeniu (rys. 4.14*a*). Liczba kroków (obiegów pętli) tego algorytmu jest równa liczbie cyfr mnożnej.

Szczególną cechą mnożenia liczb dwójkowych jest to, że właściwie brak tam operacji mnożenia przez cyfrę. Ponieważ wartością cyfry dwójkowej może być tylko 0 albo 1, więc wynikiem mnożenia przez cyfrę jest albo 0, albo wartość liczby mnożonej. Upraszcza to znacznie algorytm, **sprowadzając mnożenie liczb do ciągu operacji dodawania i przesunięcia w prawo** (rys. 4.14*b*). W stosunku do sytuacji przedstawionej na rys. 4.14*a* mamy tu o jedną operację przesunięcia więcej, zauważmy jednak, że pierwsze przesunięcie jest z punktu widzenia wyniku operacją pustą, gdyż jest wykonywane przed dodaniem do *B* jakiegokolwiek wyniku mnożenia przez cyfrę (*B* = 0). Zaletą takiego usytuowania operacji jest możliwość jednoczesnego przesuwania sumy wyników mnożenia przez mniej znaczące cyfry (gromadzącej się w bardziej znaczącej części rejestru *BC*) i wysuwania do wskaźnika *CY* kolejnych bitów (cyfr dwójkowych) mnożnej wpisanej wstępnie do rejestru *C*. Zawartość wskaźnika *CY* może być łatwo sprawdzana przez instrukcje skoków warunkowych.

Używając wprowadzonych na rys. 4.9...4.13 nazw instrukcji maszynowych możemy napisać program realizujący algorytm mnożenia liczb 8-bitowych. Załóżmy, że wartości argumentów (mnożna i mnożnik) zostały wstępnie wpisane do komórek pamięci o adresach *32000* i *32001*. Podprogram rozmieścimy w kolejnych komórkach pamięci od adresu *32002*. Wynik (16-bitowy!) mnożenia znajduje się po zakończeniu podprogramu w parze rejestrów *BC*

Adres	Instrukcja	Kod instrukcji
32002	*LD B,0*	*6,0*
32004	*LD A,(32000)*	*58,0,125*
32007	*LD C,A*	*79*
32008	*LD A,(32001)*	*58,1,125*
32011	*LD D,A*	*87*
32012	*LD E,8*	*30,8*
32014	*SRL B*	*203,56*
32016	*RR C*	*203,25*
32018	*JP NC,32024*	*210,24,125*
32021	*LD A,D*	*122*
32022	*ADD A,B*	*128*
32023	*LD B,A*	*71*
32024	*DEC E*	*29*
32025	*JP NZ,32014*	*194,14,125*
32028	*SRL B*	*203,56*
32030	*RR C*	*203,25*
32032	*RET*	*201*

Przeglądając kolumnę symbolicznie zapisanych instrukcji łatwo sprawdzić, że program dokładnie odwzorowuje sieć działań z rys. 4.14*b*. Znacznie trudniej przetłumaczyć instrukcje na odpowiadające im kody wykonywane następnie przez mikroprocesor. W praktyce jednak nigdy nie wykonujemy tej czynności ręcznie. Jeżeli piszemy program w języku wewnętrznym, to korzystamy zawsze z symbolicznego zapisu instrukcji pozostawiając ich tłumaczenie translatorowi. **Translator tłumaczący symbolicznie zapisane instrukcje na kody języka wewnętrznego nazywamy asemblerem.** Programowanie przy użyciu symbolicznych nazw instrukcji języka wewnętrznego nazywamy **programowaniem w asemblerze.**

Aby sprawdzić działanie naszego podprogramu, musimy napisać w języku *Basic* program, który wprowadzi podprogram mnożenia do pamięci, a następnie wywoła go i spowoduje wyświetlenie otrzymanego wyniku. W tym celu musimy posłużyć się instrukcjami *POKE* i *USR*

```
 5 REM PROGRAM W WERSJI DLA SPECTRUM
10 REM WPROWADZANIE PODPROGRAMU
20 FOR  I = 32002 TO 32032
30 READ  Y: POKE I,Y
40 NEXT  I
50 REM WYWOLANIE
60 INPUT  X, Y
70 POKE 32000, X: POKE 32001, Y
80 LET  Y = USR (32002)
90 PRINT  Y
'00 DATA 6,0,58,0,125,79,58,1,125,87,30,8,203,56
'10 DATA 203, 25, 210, 24, 125, 122, 128, 71, 29, 194, 14, 125
'20 DATA 203, 56, 203, 25, 201
```

W wersji programu dla mikrokomputera *Meritum* musimy zmodyfikować zakończenie podprogramu mnożenia zastępując instrukcję *RET* sekwencją

```
LD  H,B
LD  L,C
JP  2714
```

oraz dodać w programie w języku *Basic* instrukcje ładujące adres początku podprogramu do komórek *16526* i *16527*. W tym celu musimy zmienić trzy linie programu

```
20 FOR I = 32002 TO 32036
80 POKE 16526,2 : POKE 16527,125 : Y = USR(X)
'20 DATA 203,56,203,25,96,105,195,154,10
```

Ten sam algorytm mnożenia możemy wykorzystać do mnożenia liczb pamiętanych w trzech lub czterech bajtach. Jedynie program będzie nieco dłuższy (dodawanie i przesuwanie liczb wielobajtowych), a jego wykonanie zajmie znacznie więcej czasu. **Algorytm dzielenia jest bardziej skomplikowany, jednak i on daje się sprowadzić do wykonania ciągu operacji odejmowania i przesunięcia.**

Obliczanie wartości funkcji, takich jak np. sinus, kosinus lub logarytm wymaga skorzystania ze znacznie bardziej skomplikowanego aparatu matematycznego. Korzysta się tu z **rozwinięcia funkcji w nieskończone szeregi potęgowe (szereg Taylora)**. Na przykład dla funkcji trygonometrycznych są prawdziwe wzory

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} \pm \ldots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \frac{x^{2n}}{(2n)!} \pm \ldots$$

w których wykrzyknik (!) oznacza funkcję silni (por. p.2.1 pyt. 5)

Oczywiście, w praktycznych obliczeniach nie możemy zsumować nieskończonej liczby składników. Dla osiągnięcia pożądanej dokładności wystarczy na ogół uwzględnienie kilkunastu początkowych wyrazów szeregu. Czas wykonania programu obliczającego wartość pierwiastka kwadratowego mieliśmy okazję ocenić w p. 3.1.

Na zakończenie porównajmy zalety i wady programowania w języku wysokiego poziomu, takim jak *Basic*, i programowania w *asemblerze*.

Niewątpliwą zaletą języka *Basic* jest łatwość programowania wynikająca z:

— wprowadzenia jednolitych typów danych: stałych, zmiennych i tablic;

— możliwości wykonywania standardowych operacji arytmetycznych (i funkcji) na wartościach danych w zapisie dziesiętnym;

— zastąpienia bardzo licznego zbioru prymitywnych instrukcji języka wewnętrznego przez mniejszy, spójny zbiór instrukcji wykonujących typowe operacje matematyczne i organizacyjne.

Do wad zaliczymy natomiast znacznie dłuższy czas wykonania programu napisanego w języku *Basic* w porównaniu z czasem wykonania programu realizującego ten sam algorytm, lecz napisanego w języku wewnętrznym. Program napisany w języku *Basic* zajmie też na ogół więcej miejsca w pamięci. Jako przykład możemy się posłużyć operacjami związanymi z wyświetlaniem obrazu na ekranie. Wartości współrzędnych punktu ekranu nie przekraczają 255 i mieszczą się w pojedynczym bajcie — użycie do ich pamiętania zmiennych języka *Basic* absorbuje niepotrzebnie pamięć i wydłuża czas obliczeń. Dla ilustracji przypomnijmy porównanie szybkości kreślenia przekątnej ekranu przy użyciu instrukcji *DRAW* (napisanej w języku wewnętrznym) i za pomocą podprogramu napisanego w języku *Basic* (p. 2.7). Różnica jest kilkukrotna.

Pora na wnioski. Łatwość programowania w językach wysokiego poziomu okazała się zaletą dominującą. **W zastosowaniach komputerów przygniatająca większość programów pisana jest właśnie w jednym z języków wysokiego poziomu. Programowanie w języku wewnętrznym jest stosowane tylko wtedy, gdy ze względu na krytyczne ograniczenia czasu wykonania programu lub rozmiaru zajętego obszaru pamięci nie możemy wykorzystać innego języka.**

Podsumowanie

— wszystkie działania arytmetyczne mogą być w zapisie dwójkowym wyrażone jako ciąg operacji przesunięcia, dodawania i odejmowania;
— wszystkie funkcje matematyczne w języku *Basic* mogą być obliczone przez wykonanie ciągu działań arytmetycznych;
— programowanie w języku wewnętrznym (w *asemblerze*) wymaga specjalistycznych kwalifikacji i jest stosowane tylko w szczególnych zastosowaniach.

Pytania i zadania

1. Napisz w języku *Basic* program, który będzie tworzył w kolejnych komórkach pamięci jednowymiarową tablicę liczb z zakresu 0...255, wprowadzanych z klawiatury i pamiętanych, w jednym bajcie każda.
2. Napisz w języku wewnętrznym dwa podprogramy obsługujące tablicę liczb z pyt. 1. Pierwszy podprogram powinien pobierać kolejne bajty z tablicy, analogicznie jak instrukcja *READ* pobiera ze zbioru *DATA*, drugi powinien powodować rozpoczęcie pobierania od początku — analogicznie jak instrukcja *RESTORE*.
3. Napisz w języku *Basic* program obliczający wartość funkcji sinus za pomocą rozwinięcia tej funkcji w szereg potęgowy Taylora. Sprawdź, ile wyrazów trzeba obliczyć, aby wynik nie różnił się od wartości funkcji *SIN(X)* więcej niż o 0.01.
4. Czy *asembler* tłumaczący symbolicznie zapisane instrukcje na kody języka wewnętrznego jest kompilatorem czy interpretatorem?

5
Rozwiązania zadań

Rozdz. 1.1

1. Jednostka centralna mikrokomputera z klawiaturą, monitor i zasilacz.
2. Błędne: (1,2 + 2)
3. *PRINT wyrażenie*
4. *PRINT* "********", "********"

Rozdz. 1.2

1. Wyrażenia arytmetyczne (i ich szczególne przypadki: stałe i zmienne), teksty i zmienne tekstowe. Argumenty oddzielamy średnikiem lub przecinkiem (który powoduje tabulację).
2. *10 INPUT "A,B="; A,B : PRINT "A + B ="; A + B*
3. Tekst (który ma zostać wyświetlony na ekranie) i nazwy zmiennych (których wartości wprowadzamy). Tekst oddzielamy od zmiennych średnikiem, nazwy zmiennych oddzielamy od siebie przecinkami.
4. *INPUT* : wyświetla tekst *IMIE I WIEK* ? i oczekuje na wprowadzenie danych. Podajemy dane, np.: *EWA, 16 ENTER*, które zostają zapamiętane jako wartości zmiennych *X$* i *Y*. *PRINT* : wyświetla tekst *EWA* (wartość *X$*) i bezpośrednio za nim *MA 16* (wartość *Y*) *LAT*.
5. Wprowadzając puste linie o tych samych numerach.
7. Można. Polecenia *RUN*.

Rozdz. 1.3

1. Zarówno instrukcje jak i ich argumenty mikroprocesor może pobierać tylko z pamięci operacyjnej. Program zapisany w pamięci zewnętrznej musi być przed wykonaniem przepisany do pamięci operacyjnej.
3. Rozszerzyć obszar pamięci przydzielony dla pamiętania zmiennych łańcuchowych. Poleceniem *CLEAR m*, gdzie *m* określa rozmiar obszaru (np. 100, 150,...).

5 a) polecenie *CSAVE "n"*, gdzie *n* jest nazwą programu;
b) cofnij taśmę; polecenie *CLOAD? "n"*;
c) program w pamięci różni się od zapisanego na taśmie, polecenie *CLOAD? "n"* zasygnalizuje błąd;
d) polecenie *CLOAD "n"* i *RUN*, dopisana instrukcja numer 15 zniknie, gdyż na taśmie jej nie było.

Rozdz. 1.4

3 Może. Kropki. Np.: *PRINT (1.2 ↑ 3.4 − 5) ∗ 8*
4 Może. Np.: *PRINT "text"*
5 Takie same. Np.: *PRINT "TABU"; "LACJA",1,1,1*
6 Wszystkie się wykonają, możesz używać małych liter.

Rozdz. 1.5

1. *10 PRINT INK 7; PAPER 0*; "∗∗∗∗"; *INK 0; PAPER 7;* "∗∗∗∗"
lub:
10 PRINT INVERSE 1; "∗∗∗∗"; *INVERSE 0*; "∗∗∗∗"
2 Polecenie *SAVE "nazwa"*; cofnij taśmę, po czym *VERIFY "nazwa"*
3 *10 INPUT* "Kolor=" ; x
20 BORDER x
4 *10 PRINT INVERSE 1, FLASH 1; "ALARM"*
20 PRINT, "∗∗∗∗∗"
5. Polecenie *SAVE "G4"* ; polecenie *SAVE "AG"* ; polecenia *LOAD "G4"* i *RUN* ; polecenia *MERGE "AG"* i *RUN* — wykona się program *G4* i *AG* ; polecenia *LOAD "AG"* i *RUN* — wykona się tylko program *AG*. Polecenie *LOAD* usuwa z pamięci poprzedni program i wpisuje nowy, polecenie *MERGE* dopisuje nowy program zmieniając w poprzednim programie tylko linie o wspólnych numerach.

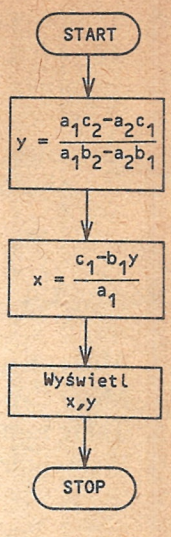
Rozdz. 2.1

1. Z bloków operacji (prostokąty), bloków badania warunków (romby), bloków początku i końca oraz strzałek.
2. Rozwiązujemy równania na papierze, np. obliczamy z pierwszego:

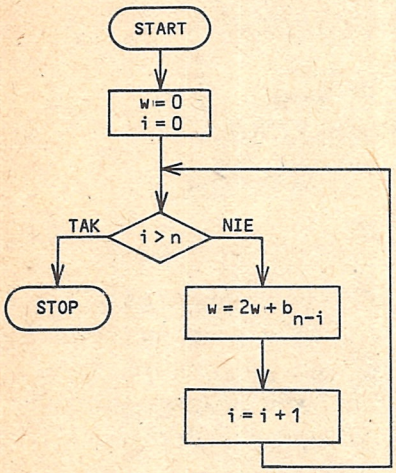
$$x = \frac{c_1 - b_1 y}{a_1} \text{ i podstawiamy do drugiego } y = \frac{a_1 c_2 - a_2 c_1}{a_1 b_2 - a_2 b_1}$$

po czym korzystając z otrzymanych wzorów rysujemy sieć działań (rys. 5.1)

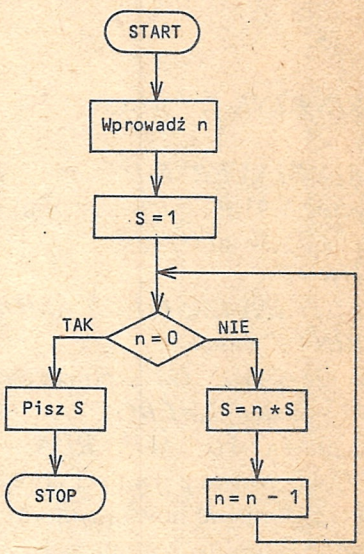
3. Algorytm jest poprawny dla każdej wartości n, choć z warunków zadania wynika, że sens mają tylko wartości dodatnie, całkowite.
4. Wzór na wartość w możemy przedstawić w postaci:
   $w = b_0 + 2\,(b_1 + 2\,(b_2 + \ldots + 2\,(b_{n-1} + 2b_n)\ldots))$
   Sieć działań przedstawiono na rys. 5.2.
5. Rys. 5.3
6. Znajdowanie największej liczby w ciągu $x_1,\ldots,x_n$



Rys. 5.1. Rozwiązanie zadania 2.1.2



Rys. 5.2. Rozwiązanie zadania 2.1.4



Rys. 5.3. Rozwiązanie zadania 2.1.5

Rozdz. 2.2

1. 1,2 — przecinek, 2^2 — nie jest to stała, lecz wyrażenie arytmetyczne zapisane niezgodnie ze składnią języka, 2.2E1.2 — niecałkowity wykładnik, + −2.2 — dwa znaki, 1/2 — wyrażenie arytmetyczne.
2. 1.25E2, 3.1415E2, 3.4E2, −340E2, 0.12845E2
3. *1A* — pierwsza cyfra, Δ — nie należy do alfabetu, *A/3* i *A$3* — drugi znak nie jest literą ani cyfrą, oś — *ś* nie należy do alfabetu łacińskiego, A_2 — indeks, *A(−1)* — nawiasy i minus.

4. Błędne: $X \uparrow 2 - 3A \uparrow 5$, $\frac{1}{2} + 3 * B$, $(A + B)\,(C + X)$,
$SIN\,(X) - COS\,(Y)$, $(A/-2) * 3$, $(3 + (A + (5 + SQR\,(X))))$

Rozdz. 2.3

1. *zm* — dowolna zmienna, *zp* — zmienna prosta, *wa*, *np*, *wk*, *wm* — dowolne wyrażenia arytmetyczne, *warunek* — relacja porównująca wartości dwóch wyrażeń arytmetycznych.
2. *10 INPUT "X="; X*
20 PRINT X↑2, X↑3
30 IF X < > 0 THEN GOTO 10
3. Rozwiązanie układu równań
10 INPUT "A1, B1, C1=" ; A1, B1, C1
20 PRINT X↑2, X↑3
*30 LET Y = (A1 * C2 − A2 * C1)/(A1 * B2 − A2 * B1)*
*40 LET X = (C1 − B1 * Y)/A1*
50 PRINT "X=" ; X, "Y="; Y
Obliczenie silni
10 INPUT "N=" ; N
20 LET S = 1
30 IF N = 0 THEN GOTO 100
*40 LET S = N * S*
50 LET N = N − 1
60 GOTO 30
100 PRINT "N!="; S
Znajdowanie największej liczby ciągu $x_1,\ldots,x_n$
100 LET M = X(1)
110 FOR I = 2 TO N
120 IF M < X(I) THEN LET M = X(I)
130 NEXT I
140 PRINT "MAX=" ; M
4. Tablicowanie funkcji e^x
10 PRINT "X" , "EXP (X)"
20 FOR X = 0 TO 1.01 STEP 0.1
30 PRINT X, EXP (X)
40 NEXT X
5. $X = 5$, $Y = 12$

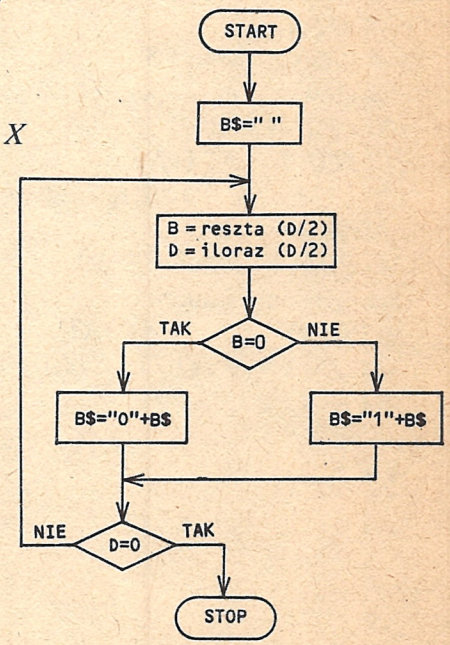
Rozdz. 2.4

2. Dodaj do programu z pyt. 3, rozdz. 2.3 instrukcje:
   10 LET N = 10
   20 DIM X (N)
   30 FOR I = 1 TO N
   40 READ X(I)
   50 NEXT I
   150 DATA 1.2, 0.125E1, 1234E−5,5, 5.34, 6.12
   160 DATA 6.12E−2, 0.611E1, 61.5E−1, 0.613E1
3. Inaczej działa funkcja *INKEY$*
4. Wartość w liczby $c_n \ldots c_1 c_0$ obliczamy jako:
   $w = c_0 + 3c_1 + \ldots + 3^n c_n =$
   $= c_0 + 3\,(c_1 + 3\,(c_2 + \ldots + 3\,(c_{n-1} + 3c_n)\ldots))$
   Sieć działań jest analogiczna do pokazanej na rys. 5.2, program podobny do programu obliczania wartości liczby dwójkowej.
5. Program analogiczny do programu z pyt. 3, rozdz. 2.3.

Rozdz. 2.5

3. Algorytm oblicza cyfry kolejno od najmniej do najbardziej znaczącej; przed wyprowadzeniem liczby należy więc zmienić kolejność cyfr. Sieć działań pokazano na rys. 5.4.

4. Podprogram wyświetlania liczby D w zapisie dwójkowym (rys. 5.4):
   1000 LET B$=""
   1010 LET X = INT (D/2)
   *1020 LET B = D − 2 * X : LET D = X*
   1030 IF B = 0 THEN GOTO 1050
   1040 LET B$="1"+B$;GOTO 1070
   1050 LET B$="0" + B$
   1070 IF D < > 0 THEN GOTO 1010
   1080 PRINT B$
   1090 RETURN
   Program wprowadzania liczb dwójkowych napisaliśmy w rozdz. 2.4, należy go tylko przerobić na podprogram. Ogólna budowa programu kalkulatora jest taka sama jak kalkulatora rzymskiego.



Rys. 5.4. Rozwiązanie zadania 2.5.3

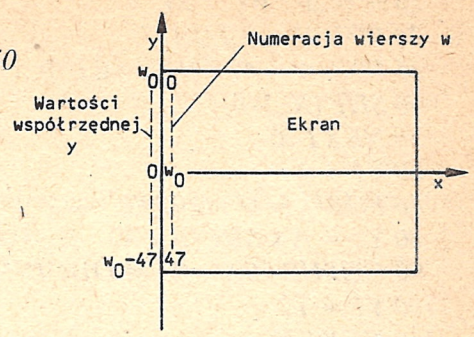
Rozdz. 2.6

2. Funkcja *TAB* (kolumna), przecinek jako separator.
5. Jeżeli umieścimy oś $0x$ układu współrzędnych w wierszu w_0 ekranu (rys. 5.5), to wzór pozwalający przeliczyć wartość współrzędnej y na numer wiersza w ma postać:

$$w = w_0 - y$$

Podprogram rysowania odcinka:

```
1000 IF Y2 < Y1 THEN GOTO 1050
1010 FOR Y = Y1 TO Y2
1020 SET (X, W0 - Y)
1030 NEXT Y
1040 RETURN
1050 FOR Y = Y2 TO Y1
1060 SET (X, W0 - Y)
1070 NEXT Y
1080 RETURN
```

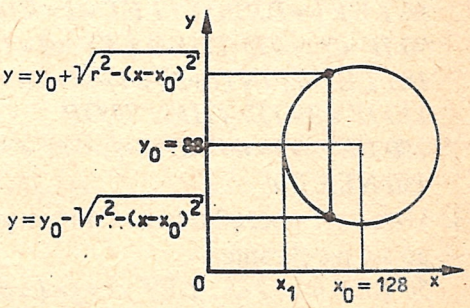


Rys. 5.5. Do zadania 2.6.5

Rozdz. 2.7

2. Żaden. Narysuje i błyskawicznie wymaże linię.
3. *10 INPUT "a,h =" ; a, h*
   20 REM lewy wierzcholek
   30 LET x = (256 − a)/2 : LET y = (176 − h)/2
   40 REM Rysowanie
   50 PLOT x, y : DRAW a,0
   60 DRAW (−a/2), h : DRAW (−a/2), (−h)
   Aby podprogram działał pewnie w każdych okolicznościach, należałoby na początku sprawdzić, czy $a = 0...255$ i $h = 0...175$.
4. Posłużymy się równaniem okręgu $(x - x_0)^2 + (y - y_0)^2 = r^2$ przekształconym do postaci $y = y_0 \pm \sqrt{r^2 - (x - x_0)^2}$. Dla kolejnych wartości x będziemy rysować pionowe cięciwy okręgu (rys. 5.6)

```
10 INPUT "r=" ; r
20 LET r2 = r ↑ 2
30 FOR x = −r TO r
40 LET y2 = r2 − (ABS x) ↑ 2
50 IF y2 < 0 THEN STOP
60 LET y = SQR (y2)
70 PLOT (128 + x), (88 − y)
80 DRAW 0, (2 * y)
180 NEXT x
```



Rys. 5.6. Do zadania 2.7.4

2. Algorytm dodawania składa się z 3 kroków:
   $CY,c \leftarrow c+f$
   $CY,b \leftarrow b+e+CY$
   $CY,a \leftarrow a+d+CY$
3. Może, np.: 01000000 (= +64)
   10000000 (= $-128 \neq 2 * 64$)

Rozdz. 4.3

2. Nie. Mikroprocesor jest niepodzielnym i nienaprawialnym elementem.
4. Za pomocą 16 bitów można zakodować adresy z zakresu od 0 do $2^{16}-1$. Maksymalna pojemność pamięci wynosi więc 2^{16}=64k=65536 komórek (bajtów).

Rozdz. 4.4

1. Instrukcje przesłań.
2. Podprogram spowoduje zapisanie 16-bitowej liczby do pary rejestrów *BC* i przesunięcia zawartości tych rejestrów w prawo:

instrukcja	kod dziesiętny
LD A, (32000)	*58,0,125*
LD C,A	*79*
LD A, (32001)	*58,1,125*
LD B,A	*71*
SRA B	*203,40*
RR C	*203,25*
RET	*201*

3. Główną trudnością programu jest umieszczenie dzielonej liczby *X* w 2 bajtach. Oznaczając bardziej znaczący bajt przez *B*, a mniej znaczący przez *C* otrzymujemy:
   $X = 256 * B + C$
   Jest to wzór identyczny ze wzorem wynikającym z definicji dzielenia z resztą liczb całkowitych *X*:256. *B* jest ilorazem, a *C* resztą tego dzielenia. Dla liczb dodatnich iloraz całkowity możemy obliczyć za pomocą operacji dzielenia i funkcji *INT*.
   Umieszczenie argumentu w komórkach pamięci o adresach *32000* i *32001* realizuje program:
   100 INPUT "X=";X
   *110 LET B=INT (x/256) : LET C=x−256 * B*
   120 POKE 32000,C : POKE 32001,B
   Napisany fragment stanowi środkową część programu. Początkowa część programu musi umieścić podprogram dzielenia przez 2 w pamięci, a koń-

cowa wywołać podprogram i wyświetlić wynik. Przykład tych czynności jest podany w rozdz. 4.5. Spróbuj jednak napisać je sam.

4. Korzystając ze znanych rozkazów możemy zapisać pierwszy składnik dodawania w rejestrze *BC*, po czym wczytywać go do rejestru *A* i dodawać kolejne bajty drugiego składnika:

instrukcja	kod dziesiętny
LD A,(32000)	*58,0,125*
LD C,A	*79*
LD A,(32001)	*58,1,125*
LD B,A	*71*
LD A,(32002)	*58,2,125*
ADD A,C	*81*
LD C,A	*79*
LD A, (32003)	*58,3,125*
ADC A,B	*88*
LD B,A	*71*
RET	*201*

Rozdz. 4.5

1. Przyjmiemy, że program tworzy tablicę na podstawie wartości zapisanych w zbiorze *DATA*. Do tablicy są przepisywane wszystkie wartości od początku zbioru aż do napotkania liczby spoza zakresu 0...255. Dla umożliwienia współpracy z podprogramami napisanymi w języku wewnętrznym adres początku tablicy zapiszemy w komórkach pamięci o adresach *32000* i *32001*.

```
10 INPUT "ADRES=";A
20 LET B=INT(A/256) : LET C=A-256*B
30 POKE 32000,C : POKE 32001,B
40 FOR I=A TO 65535
50 READ X
60 IF X<0 THEN STOP
70 IF X>255 THEN STOP
80 POKE I,X
90 NEXT I
```

2. Podprogram pobierający kolejne bajty z tablicy musi posługiwać się wskaźnikiem, którego wartością będzie adres kolejnego bajta do pobrania. Po odczytaniu bajta z tablicy wartość wskaźnika musi zostać zwiększona o 1. Początkowa wartość wskaźnika zostanie ustalona przez podprogram Restore jako równa adresowi początku tablicy. Przyjmiemy, że wskaźnik jest pamiętany w komórkach o adresach *32002* i *32003*.

Podprogram *RESTORE*

```
LD A,(32000)
LD (32002),A
LD A,(32001),
LD (32003),A
RET
```

Podprogram READ

```
LD A,(32002)
LD L,A
LD A,(32003)
LD H,A
LD B,0
LD C,(HL)
INC HL
LD A,L
LD (32002),A
LD A,H
LD (32003),A
RET
```

4. Kompilatorem.

Cena zł 229,–

Wydawnictwa Szkolne i Pedagogiczne
polecają:

P. Misiurewicz
Systemy mikrokomputerowe

A. Niederliński
Mikroprocesory, mikrokomputery, mikrosystemy

A. Niederliński
Roboty przemysłowe

Praca zbiorowa
Mikrokomputer Meritum

1341

ISBN 83-02-03589-0